Voyager 2 (version 6 release 0)

Englebert, Vincent

Publication date:
2000

Voyager 2 [1]
Reference Manual
Version 6 Release 0

V. Englebert[2]
Computer Science Department
Facultés Universitaires Notre-Dame de la Paix
Rue Grandgagnage, 21
5000 Namur
Belgium

November 27, 2000

[2] Email: vincent.englebert@info.fundp.ac.be

Preface

We believe that programs[1] like `emacs`, AutoCAD[2], Word[3] and TEX owe their success partially to the existence of a language[4] allowing the user to write macros or even programs. Indeed, such languages fill a gap between built-in functionalities and those expected by the user. This argument suffices to explain why we choose to define and to implement such a language for the `DB-MAIN` tool.

Because small steps are more secure than large ones, at the beginning – when Voyager did not exist yet– this language had to be a simple script facility for generating reports. Now, this language shares the characteristics of its big brothers and even has a name: `Voyager 2`. This last issue was the most difficult to settle!

This manual was written as a reference manual and therefore is quite concise in order to give the reader a maximum of detail economically. We are conscious that the examples are rather scarce, especially regarding the use of the repository. For this reason, this document is only a first version of what will ultimately become a series of manuals: reference manual; users's guide; and tutorial.

I thank the DB-MAIN, DB-MAIN/01, INTER-DB, PROAGEC research groups and —last but not least— the Professor J.-L. Hainaut for their support in my work.

[1] TEX is a sophisticated program designed to produce high-quality typesetting, especially for mathematical text. It was created by Donald Knuth. This manual was produced with LATEX.

[2] AutoCad and AutoLisp are trademarks of Autodesk.

[3] Word and WordBasic are trademarks of Microsoft

[4] Elisp, AutoLisp, WordBasic

Forewords

Foreword to the Version 2 Release 1

The last edition of this manual was named release 1.0. We decided to split the version number and to name each part respectively *version* and *release*. A new version introduces important modifications or significative modifications although a new release means only minor changes.

The Version 2 release 1 introduces major changes like: lexical analysis facilities[1] , new object types[2], textual properties, object removal[3], modification and meta-properties[4]in the repository.

One major change in the environment is the new console. It is no more possible to quit `DB-MAIN` in closing the `Voyager 2`'s console. This console has one disadvantage: the display is quite slow. But be sure that your program is as fast than before.

In this release, the format of the `.oxo` files has changed. So it is a good idea to recompile your former programs with the new compiler. The compiler is backward compatible.

Foreword to the Version 3 Release 0

`Voyager 2` has now the same version number as `DB-MAIN`.

Several mistakes in the reference manual have been corrected[5]. I thank Jean-Marc Hick for his help to improve the quality of the "modular programming" part with his pertinent remarks. And last but not least, I thank Richard Mairesse for his kindness — the page 51 would never have been printed without his knowledge of Postscript.

The architecture of the abstract machine has been improved. In previous releases, the abstract machine was unique and static. We can now have several abstract machines at the same time, and the number of abstract machines is not limited. This improvement allows us to call functions/procedures from other V2 programs (*cfr.* 16).

Voyager has now standard Windows dialog boxes (*cfr.* page 48).

The use of the compilers and `DB-MAIN` are now limited to people having an electronic key. That 's the price we have to pay for being famous. Without

[1] Chapter 9.
[2] Sections 11.39 and 11.38.
[3] Chapter 14.
[4] Chapter 15.
[5] I thank Philippe Thiran for his help.

the electronic key, it is impossible to run the compiler and DB-MAIN behaves in "demo" mode (the size of the repository is limited). A whole chapter explains these changes (*cfr.* 18).

The programs[1] listings have been removed from the appendices and the font has been reduced to get a smaller and more handy manual.

A β-version[2] existed during a while between the releases 2.1 and 3.0. This version allowed the use of one undocumented procedure (call_V2). This statement is now deprecated and should no more be used. (*cfr.* 16)

Foreword to the Version 4 Release 0

The abstract machine and the compilers have been translated in 32 bits. And hence, some limitations vanished. The repository has been improved in order to represent graphical properties (position, font, size, ...). The following bugs have been fixed:

- Concatenation of two empty lists.
- The documentation now describes the CharToStr function.
- The SetFlag function is fixed.

The main change is –without contest– the evolution of the repository in order to generalize it and to represent various kind of information like programs, process, procedures, etc. For this reason, we decided to attach the coll_et object-type to data_object via the DATA_COLET link and no more to the entity_type object-type. Unfortunately, Voyager 2 was not able to both take into account this generalization and to preserve the existent programs. The reason is a risk of failure when the user navigates through the old entity_colet link.

So, a query expressed as

```
entity_type: ent;
...
ENTITY_TYPE[ent]{@ENTITY_COLET:L}
```

should be translated into

```
data_object: dta;
dta2ent(DATA_OBJECT[dta]{@DATA_COLET:L with GetType(dta)=ENTITY_TYPE})
```

where dta2ent is a function that you can write yourself as:

```
function list dta2ent(list: L1 )
  entity_type: ent;
  list: L2;
{ for ent in L1 do {
    AddLast(L2,ent);
  }
  return L2;
}
```

[1] The user can find these files in the DB-MAIN distribution.

[2] This version also had some dialog boxes and was distributed to some partners.

As you understood it, the compiler will now stop if the `entity_colet` constant is encountered.

Another modification was the introduction of the first concepts to represent "process"[1] (see 95).

Foreword to the Version 5 Release 0

This version is endowed with a large number of new concepts. Processes and advanced graphical representations are now supported by `DB-MAIN`. The repository has thus considerably growed to such an extent that its representation does no more take up one sheet! Some functions have also been added (see section 7.8) since the previous version.

People have indicated some troubles when the path of the compiler contains one or several space character. This problem can be avoided with the `-quote` parameter in the command line. Unfortunately, this does not work with Windows NT.

They are now two switches in the electronic key. One to allow the programmer to port `.oxo` files to some other station with a precise electronic key; and another key to distribute them in an anonymous way (the `-Kall` option). The compiler displays the status of those switches.

The extension of the repository has introduced new keywords. The programmer should check if these words were not used as variables or functions names before. The `posx`, `posy`, and `color` attributes are now managed in a different way (see section 7.8).

Foreword to the Version 6 Release 0

This version has no great changes. Several bugs has been fixed (mainly in the libraries). Amongst the bugs, we can cite:

- the parameters of a foreign procedure.
- the assignment between processes when a process does not exists.
- the behaviour of some dialog boxes.
- the compiler now generates error messages that are Emacs-compliant.
- the size of the stack has been increased (3×)
- the `SetProperty` has been fixed (carriage return).
- the concatenation of strings (empty strings).

[1] In the large!

Contents

List of Tables

Part I

The Voyager 2 Language

Chapter 1

Preliminaries

`Voyager 2` is an imperative language with original characteristics like list primitive type with garbage collection and declarative requests to the predefined repository of the `DB-MAIN` tool. Other characteristics will be discussed further in this document. Because `Voyager 2` is similar to traditional languages like C and Pascal, we will suppose in this reference manual that the reader has a good knowledge of them.

A `Voyager 2` program is composed of three distinct sections given below:

global variables definitions
functions definitions
begin
the main body
end

The *global variables definitions* section contains the definition of all the global variables of the program. The scope of these will be the whole program as well as the functions and the procedures. Constants can also be defined in this section. The *functions definitions* section will contain the definition of all the functions and all the procedures needed by the program. Functions will not be distinguished from procedures in this document unless they are explicitly mentioned. The scope of a function is the whole program[1]. The *main body* section is the main program, *ie.* a list of instructions enclosed between the two keywords `begin` and `end`. Only the last section is mandatory in a `Voyager 2` program. As you may have guessed it, the main program corresponds to the *main* function in C and thus `Voyager 2` begins the execution there.

The *Global Variables Definitions* section contains the definition of all the global variables and all the constants of the program. This section is composed of *definition lines*, each one respects the following syntax:

⟨*definition line*⟩ ← ⟨*type*⟩ : ⟨*var-const*⟩ , ... ,⟨*var-const*⟩ ;
⟨*var-const*⟩ ← ⟨*variable*⟩ | ⟨*constant*⟩
⟨*variable*⟩ ← ⟨*identifier*⟩
⟨*constant*⟩ ← ⟨*identifier*⟩ = ⟨*expression*⟩

Types are explained in chapter 3. In a line definition, when an expression is associated with an identifier, this variable is considered as being initialized

[1] Here is a first difference with Pascal: a function f can call a function g defined afterwards.

by this expression. Each time this variable is used, its occurrence is replaced by the corresponding expression. This characteristics differs from the languages C[1] and Pascal since these last ones evaluate the expression as soon as it is found. In the `Voyager 2` language, the evaluation process is delayed until the variable is used. As a consequence, constant expression may contain identifiers and function names that are outside the scope of the expression. Unlike macros in C, constants have a type and the evaluation of the constant must match it.
Example:

Program 1.

```
integer:  s=m+c2, age;
integer:  lname=strlen(pname),m,c2;
string:  pname="Einstein";
begin
 m:=2; c2=3;
 print(s*2);
 m:= 4;
 print(s*2);
end
```

will print the values "10" ((2 + 3)*2) and "14" ((4 + 3)*2). Let us note that the evaluation of constants may return different values depending on the context.

[1]The comparison does not hold neither with the macros of the C language nor with the `const` type specifier of the C++ language

Program 2.

```
integer:  sum=a+b;
procedure foo(integer:  a)
    integer:  b;
{   b:=1;
    print(sum);
}
begin
    foo(2);
end
```

will print the value “3”.

□

The functions definitions section will contain all the functions/procedures definitions. The syntax of a function/procedure definition is fully explained in the chapter 8. Each function/procedure can be called from anywhere in the program: from a function, from a procedure or from the main body even if the call to the function/procedure is before its definition. This does not matter in `Voyager 2`.

Chapter 2

Lexical Elements

2.1 Comments

A comment in a `Voyager 2` program begins with an occurrence of the two characters `/*` not within a character or string constant and ends with the first of the occurrence of the two characters `*/`. Comments may contain any characters and may spread over several lines of the program. Comments do not have any effect on the meaning of the program you are writing.

A comment may also be any characters found after the two characters `//` in one line. There may be not have other characters between the both `//`.
Example:

```
/* Add comments to
** your programs, please !  */
begin
  x:=x+1; // and comments must be pertinent !
end
```

□

2.2 Operators

The operator tokens are divided in several groups as shown in the table 2.1.

Token class	Tokens
expression operators	+ - * / mod ++ **
	or and xor not
	< > <= >= <> =
instruction operators	:= <− << >> +> <+
separators	. , ; ( ) [ ] { }

Table 2.1: Operators and separators

Expression operators are used to build new expressions from other ones, instruction operators are a convenient way to replace classical functions by infix operators.

2.3 Identifiers

An *identifier* is a sequence of letters, digits and underscores. An identifier must begin with a letter, identifiers beginning with an underscore are reserved for keywords having a special meaning for the language. There is no restriction on the length of an identifier. Finally an identifier must be distinct of any reserved keyword (*cfr.* section 2.4) and any predefined constant name (*cfr.* section 2.5).
Example:

`factorial`	`PI_31415`	`A_B__C_`	are all valid identifiers.
`_PI`	`314_PI`	`for`	are all incorrect identifiers.
`A_Einstein`	`A_EINSTEIN`	`a_einstein`	are three distinct identifiers

□

2.4 Reserved words

Some words (*cfr.* tables 2.2 and 2.3) are reserved for the language and can not be redefined by the user.

_GetFirst	_GetNext	AddFirst	AddLast
and	as	AscToChar	begin
break	call	case	CharIsAlpha
CharIsAlphaNum	CharIsDigit	CharToAsc	CharToLower
CharToStr	CharToUpper	ClearScreen	CloseFile
continue	create	delete	do
else	end	Environment	eof
ExistFile	export	for	function
get	GetAllProperties	GetCurrentObject	GetCurrentSchema
GetFirst	GetFlag	GetLast	GetProperty
GetType	goto	halt	if
in	IsActive	IsNotNull	IsNoVoid
IsNull	IsVoid	kill	label
Length	member	mod	neof
OpenFile	or	otherwise	print
printf	procedure	read	readf
rename	repeat	return	SetFlag
SetPrintList	Setproperty	StrBuild	StrConcat
StrFindChar	StrFindSubStr	StrGetChar	StrGetSubStr
StrItos	StrLength	StrSetChar	StrStoi
StrToLower	StrToUpper	switch	TheFirst
then	TheNext	to	until
use	void	Void	while
xor			

Table 2.2: Reserved keywords.

2.5 Constants

In `Voyager 2`, *constants* are predefined variables with constant expressions. The constants names are listed in tables 2.4, 2.5, 2.6, 2.8, and 2.7.

attribute	char
cluster	co_attribute
coll_et	collection
complex_user_object	component
connection	constraint
data_object	do_attribute
document	ent_rel_type
entity_type	et_role
file	generic_object
user_object	group
integer	list
member_cst	meta_object
meta_property	owner_of_att
product	real_component
rel_type	role
schema	set_of_product
set_product_item	si_attribute
string	sub_type
system	

Table 2.3: Reserved keywords (types).

_char	_char
_file	_integer
_lambda	_list
_program	_string
ATTRIBUTE	CLUSTER
CO_ATTRIBUTE	COLL_ET
COLLECTION	COMPLEX_USER_OBJECT
COMPONENT	CONNECTION
CONSTRAINT	DATA_OBJECT
DO_ATTRIBUTE	DOCUMENT
ENT_REL_TYPE	ENTITY_TYPE
ET_ROLE	GENERIC_OBJECT
USER_OBJECT	GROUP
MEMBER_CST	META_OBJECT
META_PROPERTY	OWNER_OF_ATT
PRODUCT	REAL_COMPONENT
REL_TYPE	ROLE
SCHEMA	SI_ATTRIBUTE
SUB_TYPE	SYSTEM

Table 2.4: Constants denoting entity-types.

CLU_SUB	COLL_COLET
CONST_MEM	CONTAINS
DATA_GR	DOMAIN
ENTITY_COLET	ENTITY_CLU
ENTITY_ETR	ENTITY_SUB
GR_COMP	GR_MEM
IS_IN	MO_MP
OWNER_ATT	REAL_COMP
REL_RO	RO_ETR
SCH_COLL	SCH_DATA
SYS_MO	SYSTEM_SCH

Table 2.5: Constants denoting links.

_A	_R
_W	ARRAY_CONTAINER
ASS_GROUP	BAG_CONTAINER
BOOL_ATT	CHAR_ATT
COMP_GROUP	CON_COPY
CON_DIC	CON_GEN
CON_INTEG	CON_XTR
CON_COPY	DATE_ATT
EQ_CONSTRAINT	ETROUND
ETSHADOW	ETSQUARE
FALSE	FLOAT_ATT
HIDEPROD	INC_CONSTRAINT
INDEX_ATT	INT_MAX
INT_MIN	L_CRITERION
L_DATE	L_FREE
L_NAME	L_ROLE
L_SNAME	L_VERSION
LIST_CONTAINER	MARK1
MARK2	MARK3
MARK4	MARK5
MAX_STRING	N_CARD
NUM_ATT	OBJECT_ATT
OR_MEM_CST	PROP_CORRUPTED
PROP_NOT_FOUND	RTROUND
RTSHADOW	RTSQUARE
SCHEMA_DOMAINS	SELECT
SEQ_ATT	SET_CONTAINER
TAR_MEM_CST	TRUE
UNIQUE_ARRAY_CONTAINER	UNIQUE_LIST_CONTAINER
USER_ATT	VARCHAR_ATT

Table 2.6: Miscellaneous Constants.

atleastone	coexistence
container	creation_date
criterion	decim
disjoint	exclusive
file_desc	filename
flag	font_name
font_size	identifier
key	last_update
length	mark_plan
max_con	max_rep
mem_role	min_con
min_rep	multi
name	other
path	posx
posx2	posy
posy2	predefined
primary	recyclable
reduce	secondary
sem	short_name
stable	status
tech	text_font_name
text_font_size	total
type_object	type_of_file
type	updatable
value	version
view	where
xgrid	ygrid
zoom	

Table 2.7: Field Constants.

ERR_CALL	ERR_DIV_BY_ZERO
ERR_ERROR	ERR_FILE_CLOSE
ERR_FILE_OPEN	ERR_PATH_NOT_FOUND
ERR_PERMISSION_DENIED	

Table 2.8: Error Constants.

Chapter 3

Types

3.1 Integers

Integer type covers all the integer values from `INT_MIN` to `INT_MAX`. Integers are signed and the integer constant `INT_MIN` (resp. `INT_MAX`) is the smallest (resp. greatest) value of this type. Integer constants are signed[1] literals composed of digits 0,1,...,8,9. The integer type is named `integer`.
Examples:

1, 123, -458, -1021 are valid integer constants
+458, 3.1415, 3E+6 are not valid integer constants

□

3.2 Characters

The character type covers the whole ASCII character set from code 0 to 255. All the characters having a graphic representation have a corresponding constant in this type: the graphic representation itself enclosed between simple quotes. Otherwise characters can be represented by their ASCII value like `'^`*val*`^'`.
Examples:

```
char:  a='a', Z='Z', plus='+';
char:  bell='^7^', strange='^236^';
```

□

Some interesting non-graphic characters have a special representation illustrated in table 3.1

3.3 Strings

Strings are sequences of characters. Although the programmer must take care of details like the size of the memory block where the string is stored in Pascal

[1] The unary operator + is not allowed.

Character	Representation
backspace	'\b'
form feed	'\f'
newline	'\n'
carriage return	'\r'
tab	'\t'
'	'''

Table 3.1: Conventions about special characters.

Character	Representation
backslash \	\\
double quote "	\"
hat ^	\^
backspace	\b
form feed	\f
newline	\n
carriage return	\r
tab	\t

Table 3.2: Meta-characters used in string constants.

and C, these mechanisms are completely transparent in `Voyager 2`. Thus in this manual, the sentence "the size of the string s" means the number of characters stored in the string s. String constants are sequence of characters between double quotes. The length of a string must be less than the value found in the constant `MAX_STRING`.

Example:

The statement

```
print("Albert Einstein")
```

will produce

```
Albert Einstein
```

and

```
print("1\tone\n2\ttwo\n^51^\tthree\n")
```

will produce

```
1    one
2    two
3    three
```

□

Let us note that in string constants some characters have a special representation depicted in table 3.2. The second part of the table uses the same conventions as for the characters

With these conventions, the way the compiler interprets a string is not completely sound without some other rules:

1. Characters are examined from left to right

2. If the ^ character is followed by a sequence of digits denoting a number between 0 and 255 followed by ^, then the whole sequence is replaced by exactly one character whose the ASCII code is the mentioned number found in the string. Otherwise, the first ^ character is interpreted literally, and the interpreter scans the right part of the string *w.r.t.* these rules.

3. If the \ character is followed by a letter (λ) and if $\backslash\lambda$ does not denote a meta-character as depicted in table 3.2, then the sequence is replaced by λ. The \ character is thus removed from the string.

Example:

The instruction

```
print("^1234^55^\^\\\h");
```

will print `^12347^\h`

□

3.4 Lists

Lists are ordered collections of values. These values can be of any type (`list` included). Because lists belong to a basic type (`list`), operations on lists are often easier than in other languages like Pascal and C. Another type –`cursor`– is strongly associated to lists and will be discussed in the next section. All the operations and operators available with this type are explained here below.

A programmer can directly enter a constant list in a program simply by specifying the components of the list between brackets like that:

```
list:  lint_ext, lint_exp;
begin
  lint_exp:=[1..20];
  lint_ext:=[1,2,3,5,8,13,21];
  print(lint_exp**lint_ext);
end
```

This program will print all the common values of the two lists: “1 2 3 5 8 13”. The first list was defined *in expansion* although the second one was defined *in extension*. The syntax of list constants is:

⟨*list constant*⟩ ← “[” ⟨*list expressions*⟩ “]” | “[” ⟨*expression*⟩ “..” ⟨*expression*⟩ “]”
⟨*list expressions*⟩ ← ∅ | ⟨*expression*⟩ (“,” ⟨*expression*⟩)*

More complicated list constant expressions follow:

Examples:

```
[1,[1..fact(1)],2,[1..fact(2)],3,[1..fact(3)],4,[1..fact(4)]]
[[],[1,[]],[2,[1,[]]],[3,[2,[1,[]]]]]
[1,2,3..10,11]                          error: dots are not allowed here!
```

□

3.5 Cursors

Cursors are references to elements of lists. A cursor can either be null or be positioned. In the last case (positioned), it can be either active or passive. Let us examine the meaning of cursors in these different cases:

null cursor: the cursor is not attached to any list and is not indicating any value.

active cursor: the cursor is positioned on a value in a list, and this value can be consulted, removed, ...

passive cursor: let us suppose that there was a cursor c positioned on the value 2 of the list l =`[1,2,3]`. Just afterwards, the value 2 is removed from the list l, therefore the cursor has no more meaning and is said being *passive*. If the program consults the value indicated by this cursor c, he is getting an execution error. Although this situation looks like the *null reference*, the situation is quite different since the cursor is still attached to the list. This case will be discussed in the section 7.3.

3.6 Files

Objects of type `file` are references to files stored on disks managed by the DOS operating system. This object becomes a real reference after the call to the function `OpenFile` whose first argument is the name of the file and second argument is an integer constant. This constant indicates the mode: `_W` if the file is created for writing, `_R` if the file is opened for reading or `_A` is the file is opened for appending. Depending on the mode, the program may read or write information. Writing always occurs at the end of the file and characters are read from the *current position*. Programs must close all the opened files before leaving. More details are found in 7.4.

3.7 References

As mentioned in the section 1, `Voyager 2` is integrated in the `DB-MAIN` tool and therefore it may access to the content of the repository. The definition of the repository needs too much pages to be explained here and a whole part is devoted to the repository in part II.

The repository of `DB-MAIN` is a database built upon a network-technology DBMS with inheritance. In this model, all the relations are *one-to-many* and for this reason, relations are named *links*. Because the model is endowed with the inheritance principle, entity types are named *object-type*. And thus a link binds two object types together. The attributes of an object type are named *fields* in our model. An instance of an entity type is named *object* or *reference*. The following table summarizes the equivalence between these concepts:

ER-schema	↔ `Voyager 2`
entity-type	↔ object-type
entity	↔ object, reference
attribute	↔ field
relation	↔ link

To each object type present in the definition of the repository of DB-MAIN, there exists a type in Voyager 2. For instance, the type group corresponds to the "group" object type. Variables or expressions of this type can either be references to an object of this object-type, or can be void (a special value denoting *nothing*), or can be not valid.

If a variable is a reference to an object, then we can get the value of a field with the "." operator. For instance, the following program prints the name of the object referenced by the variable *ent*:

```
entity_type:  ent;
begin
...
print(ent.name);
...
end
```

Expressions composed with the "." operator may also occur in the left hand side part of an assignment like in the following example:

```
entity_type:  ent;
begin
...
ent.name:="CUSTOMERS";
...
end
```

The right hand part of the "." operator is in fact an integer value identifying the field among all the others. In this example, name is a predefined integer constant.

Chapter 4

Expressions

Expressions are classified into several classes depending on the type returned by the evaluation process. Some expressions are untyped mainly due to access to the repository and to lists, for these particular cases, the type verification is delayed until the execution time. The first subsection discuss the operators used in expressions. Next subsections treat operators and functions provided by the language for each type.

4.1 Precedence and associativity of operators

Each expression operator in `Voyager 2` has a precedence level and a rule of associativity. Where parentheses do not explicitly indicate the grouping of operands with operators, the operands are grouped with the operators having higher precedence. If two operators have the same precedence, there are grouped following the associativity rule (left/right associativity). The table 4.1 defines the precedence and associativity rules of each operator.
Example:

Following complex expressions may be reduced as follows with the precedence/associativity rules:

Original expression	Equivalent expression
`a+b*c`	`a+(b*c)`
`a=not b and c or d`	`a=(((not b) and c) or d)`
`a.length > 10 = 1`	`((a.length)>10)=1`

□

4.2 Arithmetic Expressions

Operators +[1], -, *, /, `mod`, `and`, `or`, <, >, <=, >=, <>, =, `not` all require integer expressions as operand. Their operands are fully evaluated before their own evaluation but the order is let unspecified. The definition of +, -, *, / and `mod` is respectively addition, subtraction, multiplication, division and remainder.

[1]The + operator is overloaded in order to behave like the `StrConcat` function with strings.

Token	Operator	Class	Associates	Operands[a]
`not`	logical not	prefix	no	i
`-`	unary minus	unary	no	i
`*`	multiplicative	binary	left	i
`/`	division	binary	left	i
`mod`	modulo	binary	left	i
`**`	list intersection	binary	left	l
`+`	addition	binary	left	i,s
`++`	difference	binary	left	i
`*`	list concatenation	binary	left	l
`<`	less than	binary	left	i, c, s
`>`	greater than	binary	left	i, c, s
`<=`	less than or equal	binary	left	i, c, s
`>=`	greater than or equal	binary	left	i, c, s
`<>`	different	binary	left	i, c, s
`=`	equal	binary	left	i, c, s
`and`	logical and	binary	left	i
`or`	logical or	binary	left	i
`xor`	logical xor	binary	left	i
`:==`	functional assignment	binary	right	any
`,`	separator	binary	left	any

Lines separate operators following their precedence. One line separates two groups of operators and each operator inside one group have the same precedence. If a group is above another one, then the precedence of its operator is higher than the other group. For instance: prec(*)>prec(+).

[a]Operands must always be of the same type. Following letters denote expected types by previous operators: `l`: list, `i`: integer, `c`: char, `s`: string, `any`: any type.

Table 4.1: Operators: Precedence and Associativity rules

When the divisor is zero (x/y and $y = 0$), then the result is 0 and the error register is set to `DIV_BY_ZERO`. For the other ones, the following table gives a formal definition:

$$
\begin{aligned}
a > b &\text{ returns : if } a > b \text{ then 1 else 0} \\
a < b &\text{ returns : if } a < b \text{ then 1 else 0} \\
a <= b &\text{ returns : if } a \leq b \text{ then 1 else 0} \\
a >= b &\text{ returns : if } a \geq b \text{ then 1 else 0} \\
a = b &\text{ returns : if } a = b \text{ then 1 else 0} \\
a <> b &\text{ returns : if } a \neq b \text{ then 1 else 0} \\
\texttt{not}(a) &\text{ returns if } a = 0 \text{ then 1 else 0}
\end{aligned}
$$

4.3 Reference Expression

The word "reference" groups several types together and do not make distinction among them. Let us examine the following line:

```
attribute: att;
```

att is a variable that may reference an object in the repository. We will see in chapter 13 that the following statement:

```
att:=GetFirst(attribute[a]{TRUE})
```

puts into the variable *att* the reference to the first attribute found in the repository of `DB-MAIN`. This variable can be used to consult or modify properties of the object:
Example:

```
print(att.name);
att.name:="FIRST-NAME";
```

□

The left part of the "`.`" separator must be an identifier (global/local variable, parameter) denoting an object. The left part must be a field name valid for the object specified in the right part. The right expression must be either an integer expression or a string. String fields will be explained later in chapter 15.

4.4 Functional Assignment

This operator behaves like the assignment operator (`:=`) except that the left-hand-value is let on the stack afterwards. This operator is noted "`:==`".
Example:

```
integer:  a,b;
string:  s;
begin
    a:=(b:==0);
    if ((s:==read(_string))="Hello") then {
       print(" World!");
    }
end
```

□

The first statement initializes the two variables `a` and `b` with the value 0. The second statement reads one string from the console, puts it into the variable `s` and then compares this string with the string "World!".

More explanations can be found in the section 6.1 at page 31.

Chapter 5

List Expressions

5.1 Overview

Lists in `Voyager 2` have no similar counterparts in Pascal and C. As explained in the subsection 3.4, lists are ordered collections of values. A list has an existence which is not directly linked to the scope of variables representing it. Values in lists may be of any type, even `list`, `cursor`, ... A list exists in memory until the program can no more use the values contained in this list, and the programmer does not have to care about the memory management. Let us remember that values can be get through cursors or variables of type `list`.

Because lists in `Voyager 2` are quite different from lists in other languages like Pascal, C and Lisp, some definitions are necessary.

Definition 5.1 (List) *Let l be a variable denoting a list of values $v_1, \ldots, v_n$, we write $[v_1, \ldots, v_n]$ the content of this list.*

Definition 5.2 (Ghost) *We define $\bullet$ (a ghost) a special value having no meaning in `Voyager 2`. This value may belong to lists.*

Ghosts are invisible and therefore useless, but they will be used in list's graphical representation and in explanations.

Definition 5.3 (∂) *We define a unary operator ∂. Let l be a list, then ∂l returns the list l from where all the ghosts have been removed.*

In other words, all the $\bullet$ values are removed from the list. Then $\partial[\bullet, 1, 2, [3, \bullet], \bullet] = [1, 2, [3]]$.

Definition 5.4 (list equality, $=$) *$l_1 = l_2$ iff $\partial l_1 \equiv \partial l_2$ where the $\equiv$ operator has the usual meaning. This means that ghosts are not pertinent to compare lists in `Voyager 2`.*

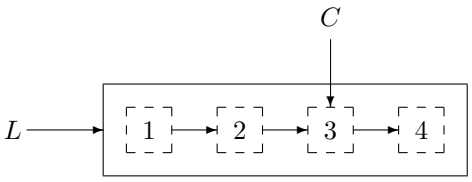
The *list equality* is the usual way to consider the equality between lists for the programmer.

We associate a graphical representation to the lists, to the variables of type `list` and to the cursors in order to make easier the explanations. A list is represented by a rectangle containing values linked by arrows. Values are represented

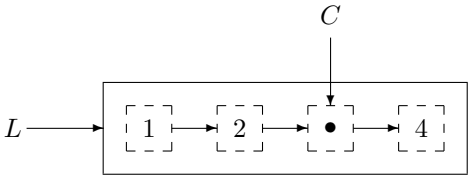
by dashed boxes. A variable v of type list is represented by an arrow towards the graphical representation of the list. A cursor c pointing to a value inside the list is denoted by an arrow towards this value. Let us consider the following program:

```
1:  list:  L;
2:  cursor:  C;
3:  begin
4:     L:=[1..4];
5:     attach C to L;
6:     C >> 2;
7:  end
```

At line 4, the list L is assigned to the list $[1, 2, 3, 4]$, at line 5, the cursor C is attached to the list L and therefore is indicating the first value of L [1]. The instruction at line 6 moves the cursor two elements forward, the cursor C is now indicating the value 3. The graphical representation of the state at line 7 is:



The instruction `kill(C)` destroys the value under the cursor. So, the instruction `kill(C);` will destroy the value 3 in L. We represent this action by replacing the value 3 in L by the special value: $\bullet$. The cursor is still attached to the list L but it is now impossible to consult the value under C or to replace this value by another one. The graphic becomes:

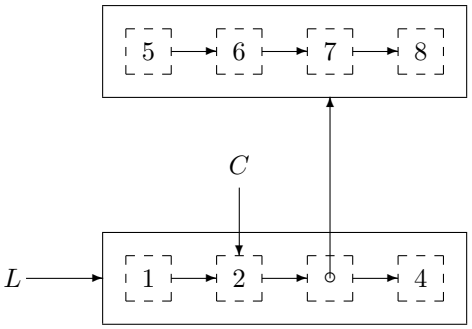


With respect to the definition 5.4, the following property holds:

$$L = [1, 2, 4]$$

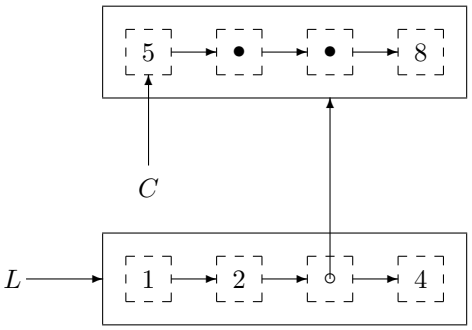
Let us suppose that the next sequence is executed right now: '`C <<; C+> [5..8]`", the cursor C is moved one element backward and the list $[5, 6, 7, 8]$ is inserted just after C. The graphic becomes:

[1] This is a convention. Newly attached cursors always point to the first element. If the list is empty ($L = []$) then the cursor is said to be null and has the special value `void`.



Let us remark that the ghost value has disappeared. The reason is simple: the cell containing the ghost value was no more referenced. So it was become safe to suppress it. From the user's point of view, it was impossible to detect the presence of the ghost value after the execution of the instruction "`C <<;`".

The value under a cursor can be consulted with the function `get`. We will use this function to attach C to the newly created list with the sequence of instructions: "`C>>; attach C to get(C);`". The cursor is now indicating the first element of the new list: 5. The following sequence will destroy the two elements 2 and 3 of this list: "`C>>; kill(C); C>>; kill(C); C>>;`", C is now indicating the last value: 8. This list is now equivalent to the list $[5, 8]$ and for this reason if the cursor C is moved one element backward, "`C <<`", one finds the value 5 under C as it is illustrated by the graphic:



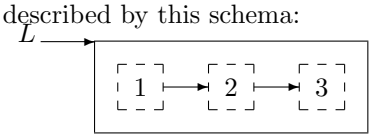
Although the general rule in `Voyager 2` for passing values to functions is by value, list objects are always passed by address[1]. Let us examine the meaning of the following program:

```
1:  list:  L,R;
2:  procedure RemFirst(list:  arg)
3:     cursor:  C;
4:  {  attach C to arg;
5:     kill(C);
6:  }
7:  begin
8:     L:=[1,2,3];
9:     R:=RemFirst(L);
```

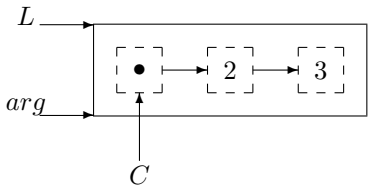
[1]Except some cases like the ++ operator and other functions.

```
10:    print([L,R]);
11: end
```

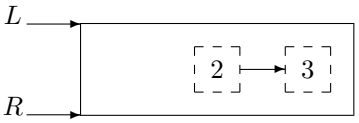
After line 8, the state is described by this schema:



at line 6, when the first element has been deleted and the `RemFirst` function has been called:



and finally at line 10:



and the result printed on the console will be:

```
[[2,3],[2,3]]
```

With respect to this principle, lists can be built inside a function and returned to the global environment. The following program is a good example of what happens when a list is returned from a function:

```
list:  L;
function list foo()
    list:  local;
{   local:=[1..3];
    return local;
}
begin
    L:=foo();
    print(L);
end
```

Because the list `[1,2,3]` is first referenced by the variable *local*, next by an intermediate element placed on the stack and finally referenced by the global variable L, the list was not destroyed when the function finished.

5.2 Operations Definitions

Several operators exist specifically for the lists. A description of each one follows.

5.2.1 Concatenation

The infixed operator ++ takes two distinct lists and returns the concatenation of both. Let us note that the arguments are detached of their body after execution. For instance, let us suppose that the cursor C is attached to the list $L1$ and that the instruction $R := L1 + +L2$ is performed right now! Then the cursor C is now attached to the list R and no more to $L1$ whose the value is the empty list [ ].

function list: r ++ (list: $l1$, list: $l2$)

Precondition. lists $l1$ and $l2$ are two list expressions **denoting distinct lists**.

Postcondition. lists $l1$ and $l2$ are now empty. Let us suppose that list $l1$ is $[v_1, \ldots, v_n]$ and $l2$ is $[w_1, \ldots, w_m]$, then the result will be a new list: $[v_1, \ldots, v_n, w_1, \ldots, w_m]$. After the call, the following property holds: $l1 = [\,]$ and $l2 = [\,]$.

The following examples show what is the effect of this operator:
Examples:

```
[1,2,3]++[4,5,6] → [1,2,3,4,5,6]
L1:=[['a',1]]; L1:=L1++[['b',2]] → [['a',1],[['b',2]]]
L2:=L1++L1; → error !
L2:=L1; L3:=L1++L2; → error !
```

□

5.2.2 Intersection

The infixed operator ** is used between two lists to compute all the common elements. There is no restrictions on the arguments of this operator.

function list: r ** (list: $l1$, list: $l2$)

Precondition. $l1$ and $l2$ are two lists. The type of the items stored in both lists may not be identical.

Postcondition. r is the list of all the values common to lists $l1$ and $l2$. If one object is present in both lists but with different types (one super-type[1] and one sub-type[2] for instance), they are considered as distinct. The order of the returned list is left unspecified.

The following instances show the power of this operator:

```
[1,1,2,4]**[1,5,2]→[1,2]
```

[1] For instance:`data_object`
[2] For instance: `ent_rel_type`

```
[[1,2],[3,4],'b',7]**['a',7,[3,4],["ab",GetCurrentSchema()]]
→[[3,4],7]
l1:=[1,2,1,3,'a','a','b'];
l1**l1→[1,2,3,'a','b']
```

5.2.3 Insertion

To insert values in lists, several methods have already been presented. But no one is as general than the new operators `+>` and `<+`. These operators are infixed. For each one, the left hand operand must be an expression of type `cursor` and the right hand operand may be any expression that can be inserted in a list. The effect of the first (resp. second) operator is to insert the result of the right hand expression just after (resp. before) the value designated by the cursor specified in the left operand.

In order to remove any ambiguity, we give here the formal definition of these two operators.

Let us analyse the effect of the following instruction:

$$C{+>}\ E$$

where C is any expression of type `cursor` and E is an expression.

1. C **is attached to a list** L**.**
   a. **The cursor** C **is null.** Then the value of E is inserted as being the first element of the list L.
   b. **The cursor** C **is not null.** Then the value of E is inserted just after the value designated by the cursor C.
2. C **is not attached to a list.** The instruction fails as well as the program. This is an error of the programmer.

For the other operator $<+$, the effect of the following instruction

$$C\ {<+}E$$

will be:

1. C **is attached to a list** L**.**
   a. **The cursor** C **is null.** Then the value of E is inserted as being the last element of the list L.
   b. **The cursor** C **is not null.** Then the value of E is inserted just before the value designated by the cursor C.
2. C **is not attached to a list.** The instruction fails as well as the program. This is an error of the programmer.

In all the cases, the cursor C is unchanged and is still designating the same value as before the call of the instruction.

5.2.4 Miscellaneous

function any: r `get` (cursor: c)

Precondition. `IsVoid(c)<>TRUE`

Postcondition. r is the value pointed by the cursor c.
on error: The program is interrupted and an error message is displayed.

Chapter 6

Statements

Each statement must be terminated by one semi-colon except for compound instructions where this character is optional. The empty statement is not allowed in `Voyager 2`, however a compound statement may be empty.

6.1 Assignment

Assignment statements must respect the following syntax:

⟨*assignment-inst*⟩ ← ⟨*lhs*⟩ := ⟨*rhs*⟩
⟨*rhs*⟩ ← ⟨*expression*⟩
⟨*lhs*⟩ ← ⟨*variable*⟩ | ⟨*variable*⟩.⟨*field*⟩
⟨*field*⟩ ← ⟨*expression*⟩

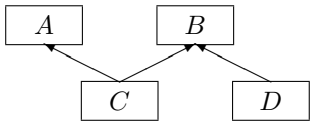
The *rhs*-expression must have a type compatible with the type of the *lhs*-expression. If the *lhs*-expression has a field, then the evaluation of the field must return an integer value. Fields are specific to variables denoting a reference to a repository's object and usually, the user will use a predefined constant (*cfr.* 2.6) in place of complex expression. When this instruction is executed, the *rhs*-expression is first computed and the result is then assigned to the *lhs*-expression.
Example:

```
int:  a,b;
string:  s;
begin
a:=1;
b:=a*2;
s:="Rob Roy";
end
```

□

But the assignment instruction does much more work than it is explained so far. Indeed, when inheritance is involved during the assignment, this instruction is able to solve automatically the ambiguity —dynamic type casting. Let us

suppose that we have the following schema representing the inheritance between the object types[1]:



and the following program:

```
A: a;
B: b;
C: c;
begin
    ...
    b:=c;
    a:=b;
    ...
end
```

The last assignment is not trivial since the object referenced by `b` could be either of type `C` or `D`. But the assignment is able to find the correct path between the type of `a` and the type of the object referenced by b[2].

If the assignment fails — types are not compatible — then the program is aborted and the conflicting types are displayed in the console.

The dynamic type casting performed by the assignment is not a general rule in `Voyager 2`. Therefore, unless it is explicitly mentioned[3], types must always be exactly identical. For instance, each time you define a new function, arguments and parameters must always have the same type. For this reason the following program is wrong:

```
C: c;
procedure foo(A:a){
    ...
}
begin
    ...
    foo(c);
end
```

The only way to pass the value `c` to the function is by using an explicit assignment like that:

```
C: c;
A: a;
procedure foo(A:a){
```

[1]The example is not a part of the real schema.

[2]Let us remark here, that the type of an object referenced by a variable may be different of the type of the variable! For instance, just after the first assignment, the type of the object referenced by `b` is `C` although the type of the variable `b` is `B`.

[3]The dynamic type casting is applied for the arguments of the function `create`.

```
    ...
}
begin
    ...
    a:=c;
    foo(a);
end
```

Often a suite of assignment is prone to be optimized. Although performances are not critical for the `Voyager 2` programmer, strings may slow down some programs like parsers. For instance, the following scheme is often observed in parsers:

```
while neof(f) do {
    s:=read(_string) ;
    if s="begin" then
       ...
end
```

The scanned string will be put twice on the stack. The function `read` will read the string from the file and place it on the stack in order to put it into the variable `s`. And the next instruction will place the value of `s` on the stack. Conclusion: we put the value we have just removed before! One obvious optimization is to not remove the value from the stack. This optimization can be achieved by the programmer in using the function assignment operator (`:==`) defined in the section 4.4.

6.2 Selection Statement

Selection statements direct the flow of control depending on the value of an expression.

6.2.1 The `if-then` Statement

The `if-then` statement executes a list of instructions if the evaluation of the condition is different from 0. The syntax is:

⟨*if-then-statement*⟩ ← `if` ⟨*condition*⟩ `then` { ⟨*list-instruction*⟩ }

The evaluation of the expression *condition* must return an integer value d. If the value d is nonzero then the list of instructions *list-instruction* is executed.
Example:

```
if n=0 then { n:=1; }
```

□

6.2.2 The `if-then-else` Statement

The `if-then-else` statement executes a list of instructions among the candidates *success* and *failure* depending on the evaluation of the expression *condition*. The evaluation of this expression must return an integer value.

⟨*if-then-else-statement*⟩ ← `if` ⟨*condition*⟩ `then` ⟨*success*⟩ `else` ⟨*failure*⟩
⟨*success*⟩ ← { ⟨*list-instruction*⟩ }
⟨*failure*⟩ ← { ⟨*list-instruction*⟩ }

The evaluation of the expression *condition* must return an integer value (d). The flow of control is directed to the list of instructions *success* (resp. *failure*) if the evaluation of the expression *condition* is nonzero (resp. *zero*).
Example:

```
if m<n then {
    v:=m;
} else {
    v:=n;
}
```

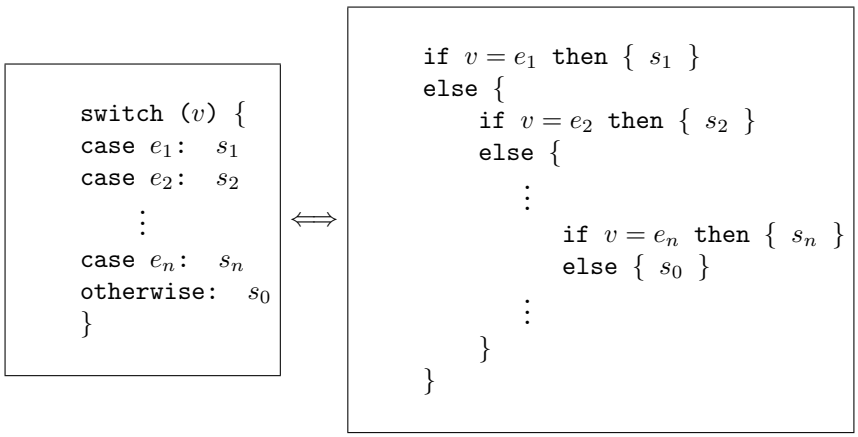
□

6.2.3 The `switch` Statement

The `switch` statement chooses one of several flows of control depending upon a criterion. The criterion must be either a variable or a variable with a field whose the type must be compatible with respect to the "=" operator with the values found in the `case` statements. Its syntax is:

⟨*switch-statement*⟩ ← `switch (` ⟨*variable*⟩ `)`
{ ⟨*case-list*⟩ ⟨*default*⟩ }
⟨*case-list*⟩ ← ∅ | ⟨*case-list*⟩ ⟨*case-sttmt*⟩
⟨*case-sttmt*⟩ ← `case` ⟨*expression*⟩ `:` ⟨*list-instruction*⟩
⟨*default*⟩ ← ∅ | `otherwise :` ⟨*list-instruction*⟩

The meaning of such a statement can be described by another equivalent `if-then-else` statement as it is showed below:



If the *default* clause is not present, then just consider that the s_0 list is empty. This translation of the `switch` statement is in no way an explanation of the compilation process, so the order of evaluation of the e_i expressions is

not garanteed by the language `Voyager 2`. Therefore, expressions for which the evaluation has a side effect are discouraged since the semantics is unspecified.

When the `switch` statement is executed the value of the variable is compared to each expression e_i until values are equal. Once this condition is satisfied, the respective list of instructions is executed. If all tests fail, then no instruction is executed unless the *default* case is present and then its list of instructions is executed.

Example:

```
switch (letter) {
case 'B':
    print("Belgium");
    print(" (Belgique)");
case 'F':
    print("France");
case 'S':
    print("Spain");
    print(" (Espagne)");
otherwise:
    print("I don't know!");
}
```

□

6.3 Iteration Statement

Iteration statements are the `while`, `repeat` and `for` instructions.

6.3.1 The `while` Statement

The `while` statement has the following syntax:

⟨*while-statement*⟩ ← `while` ⟨*condition*⟩ `do` { ⟨*body*⟩ }
⟨*condition*⟩ ← ⟨*expression*⟩
⟨*body*⟩ ← ⟨*list-instruction*⟩

The evaluation of the *condition* must be an integer value. While the evaluation of this expression will be nonzero, the *body* will be executed. The iteration stops when the evaluation of the *condition* returns the value 0.

Example:

```
f:=1;
while n>0 do {
    f:=f*n;
    n:=n-1;
}
```

□

6.3.2 The `repeat` Statement

The `repeat` statement has the following syntax:

⟨*repeat-statement*⟩ ← `repeat` { ⟨*body*⟩ } `until` ⟨*condition*⟩
⟨*body*⟩ ← ⟨*list-instruction*⟩
⟨*condition*⟩ ← ⟨*expression*⟩

The evaluation of the expression *condition* must return an integer value. The *body* is executed until the evaluation of the *condition* returns a nonzero value.
Example:

```
n:=read(_integer);
repeat {
    n:=n-1; } until n=0;
```

□

6.3.3 The `for` Statement

The `for` statement is probably the most usual instruction for doing iterations among a set of values. The original feature of this statement is certainly the iteration through elements of lists. This characteristic allows it to be used to visit references coming from the evaluation of a request. Its syntax is:

⟨*for-statement*⟩ ← `for` ⟨*iterator*⟩ `in` ⟨*list*⟩ `do` { ⟨*body*⟩ }
⟨*iterator*⟩ ← ⟨*variable*⟩
⟨*list*⟩ ← ⟨*expression*⟩
⟨*body*⟩ ← ⟨*list-instruction*⟩

The evaluation of the expression *list* must return a value of type `list`. Moreover, each element of this list must be of the same type as the variable *iterator*. If the list is empty, this statement has no effect except the evaluation of the *list*. Otherwise, the variable *iterator* is instantiated with the first value found in the list, and the *body* is executed. Afterwards, we try to iterate through the suffix of this list until the prefix becomes empty. When the `for` statement is completed, the variable *iterator* is instantiated with the last value found in the list (the value is unspecified if the list is empty).

Let us note that one do not tell neither how nor when the list is evaluated! So we recommend to avoid any instruction that could have a side-effect on the evaluation of the list. The following examples show very dangerous programs:
Examples:

```
for i in [1..n] do {
    print(i);
    n:=n+1; }

b:=1;
for i in [a,b] do {
    b:=2;
}

l:=[1..10];
```

```
for i in l do{
    l:=l++[11];
}
```

□

Here follow some correct use of the `for` instruction.

Example:

```
for i in [1..5+5] do {
    print(i);
}


for data in DATA_OBJECT[data]{@SCH_DATA:[sch]
                              with data.name="Foo"} do {
    print(data.short_name);
    for gr in GROUP[gr1]{@DATA_GR:[data]} do {
        print(gr.name);
    }
}

for c in ['a','b']++['e','f'] do {
    print(CharToUpper(c));
}

for my_list in [[1,2],[3,4],[5]] do {
    print(my_list);
}
```

□

6.3.4 The `goto` Statement

The `goto` statement directs the flow of control to a statement labeled by an identifier. If this instruction is used inside a function, the flow of control can not go out the body of the function. In the same way, the control flow can not be directed from the main body to the inside of a function. See 6.3.5 for a detailed example.

⟨*goto-statement*⟩ ← `goto` ⟨*identifier*⟩

6.3.5 The `label` Statement

The `label` statement is used to put a label in front of a statement. The syntax is:

⟨*label-statement*⟩ ← `label` ⟨*identifier*⟩

Example:

```
i:=0;
label loop;
if i<10 then {
```

```
        i:=i+1;
        goto loop;
    }
```

□

6.3.6 The `break` Statement

The `break` instruction can only be used in `for-in-do`, `while` and `repeat` instructions. For the `while` and `repeat` instructions, the effect of this instruction is equivalent to a `goto` instruction to a label put just after the `while`/`repeat` instruction. This "breaks" the loop.

For the `for-in-do` instruction, the explanation depends on the list used for the iteration. If the list expression is a predicative query, then the effect of the `break` instruction is to skip to the brother of the father of the current item. Let us consider the following example:

Example:

```
1:  owner_of_att:  o,o1,o2,o3;
2:  attribute:  a;
                 ⋮
3:
4:  for a in ATTRIBUTE[a]{@OWNER_ATT:[o1,o2,o3]} do {
5:     if GetFirst(OWNER_OF_ATT[o]{OWNER_ATT:[a]})=o1
6:     then { break; }
7:     print(a.name);
8:  }
```

□

where $o1$ owns three attributes: $a1$, $a2$ and $a3$; and $o2$ owns two attributes: $b1$ and $b2$. Then the first attribute at line 5 will be $a1$. Since the owner of $a1$ is $o1$, the test succeeds and the `break` instruction is called. Its effect will be to skip all the sons of the $o1$ owner and to go directly to the next owner: $o2$. The program will thus print: $b1$, $b2$, ...

If the list expression is not a request, then the `break` instruction will break the loop and will continue with the instruction following the `for-in-do` instruction.

6.3.7 The `continue` Statement

The `continue` instruction can only be called inside `for-do-in`, `while` and `repeat` instructions. In the `while`/`repeat` instructions, `continue` will skip the rest of the instruction's body and causes the reevaluation of condition expression.

If the instruction is used inside the `for-do-in` instruction with a request as list-expression, then its effect will be different. Let us get a look at the followiing example:

Example:

```
owner_of_att:  o,o1,o2,o3;
attribute:  a;
          ⋮
for a in ATTRIBUTE[a]{@OWNER_ATT:[o1,o2,o3]} do {
    if a=a2
    then { continue; }
    print(a.name);
}
```

□

where the context is the same as in 6.3.6. When the current item becomes $a2$, then the `continue` instruction is called. The instruction will skip the rest of the body and will look for the brother of the current element, $a2$, which is $a3$. If $a2$ were the last son of the owner, $o1$, then the `for-do-in` instruction will terminate the processing the $o1$'s sons and go to the next owner: $o2$.

If the list expression is not a request, then the continue expression will just skip the rest of the body for the current element and will process the next value in the list.

6.3.8 The `halt` Statement

The `halt` instruction can be called anywhere in the program where an instruction is expected. This instruction will stop the program. As when your program terminates, this instruction will not close your opened files!

Chapter 7

Operations

Operations are statements having the form of a call to a predefined procedure/function. Because their syntax has already been defined, we have isolated them from other statements.

7.1 Operations on Characters

function integer: d `CharIsDigit` (char: c)

Precondition. $\emptyset$

Postcondition. $d \neq 0$ if $c \in \{0, \ldots, 9\}$ and 0 otherwise.

function integer: d `CharIsAlpha` (char: c)

Precondition. $\emptyset$

Postcondition. $d \neq 0$ if $c \in \{a, \ldots, z, A, \ldots, Z\}$ and 0 otherwise.

function integer: d `CharIsAlphaNum` (char: c)

Precondition. $\emptyset$

Postcondition. $d \neq 0$ if $c \in \{0, \ldots, 9, a, \ldots, z, A, \ldots, Z\}$ and 0 otherwise.

function string: s `CharToStr` (char: c)

Precondition. $\emptyset$

Postcondition. s is a string composed of the c character.

function char: c' `CharToUpper` (char: c)

Precondition. $\emptyset$

Postcondition. if $c \in \{a, \ldots, z\}$ then c' is the respective upper case letter. All other characters are left unchanged.

function char c' `CharToLower` (char c)

Precondition. $\emptyset$

Postcondition. if $c \in \{A, \ldots, Z\}$ then c' is the respective lower case letter. All other characters are left unchanged.

function char: c `AscToChar` (integer: d)

Precondition. $0 \leq d \leq 255$

Postcondition. Character c has the ASCII code: d.
on error: $c =$`^0^`.

function integer: d `CharToAsc` (char: c)

Precondition. $\emptyset$

Postcondition. d is the ASCII code of the character c.

7.2 Operations on Strings

We will define here some definitions to make easier the explanations that follow. First of all, let s denote a string and d a positive number. Then we note s_d the d^{th} character of the string s, and $s_{d\rightarrow}$ the suffix of the string s starting at the position d (included) in the string and $s_{d\rightarrow d+l}$ the substring comprised between positions d and $d+l$ where l is a positive number such that $d+l$ does not exceed the length of the string. Let us remember a last detail: the first character of a string is placed at the position 0, and thus if l is the length of s, the last character is placed at the position $l-1$. The following operations are safe with respect to two criteria:

- The program can never write a character outside strings.
- The program can never place the null character inside a string[1].

This is a valuable guaranty against frequent bugs that C and Pascal programmers certainly know.

function string: s `StrBuild` (integer: d)

Precondition. $d \geq 0$ and $d \leq$`MAX_STRING`

Postcondition. s is a string composed of d space characters (`' '`).
on error: s is the empty string.

function string: s `StrConcat` (string: s_1, string: s_2)

Precondition. $\emptyset$

Postcondition. This function appends the string s_2 at the end of s_1 and the result is stored in s. The length of the resulting string is `StrLength(`s_1`)+StrLength(`s_2`)`. The infix operator "+" can also be used in place of the `StrConcat` function.

[1] By convention, the null character ends strings. Therefore such a possibility is troubling the memory manager.

function integer: r `StrFindChar` (string: s, integer: d, char: c)

Precondition. $0 \leq d <$ `StrLength`(s).

Postcondition. If $r \geq 0$ then $s_r = c$ and $\forall i,\ d \leq i < r :\ s_i \neq c$.
Otherwise if $r = -1$ then $\forall i,\ d \leq i <$ `StrLength`$(s) :\ s_i \neq c$.
on error: $r = -1$.

function integer: r `StrFindSubStr` (string: s, integer: d, string: t)

Precondition. $0 \leq d <$ `StrLength`(s).

Postcondition. If $r \geq 0$ then t is a prefix of $s_{d+r\rightarrow}$. Otherwise if $r = -1$ then $\forall i \geq 0$, t never is a prefix of $s_{d+i\rightarrow}$.
on error: $r = -1$

function char: c `StrGetChar` (string: s, integer: d)

Precondition. $0 \leq d <$ `StrLength`(s).

Postcondition. $c = s_d$.
on error: $c =$`'^0^'`

function string: r `StrGetSubStr` (string: s, integer: d, integer: l)

Precondition. $0 \leq d <$`StrLength`$(s)\ \wedge\ 0 \leq l \leq$`StrLength`$(s) - d$

Postcondition. $r = s_{d\rightarrow d+l}$
on error: if $d < 0$ then $d \leftarrow 0$; if $d \geq$`StrLength`(s) then $d \leftarrow$`StrLength`(s); if $l < 0$ then $l \leftarrow 0$; if $l >$`StrLength`$(s) - d$ then the function will consider that $l -$`StrLength`$(s) + d$ space characters are added at the end of the string s.

function string: s `StrItos` (integer: d)

Precondition. $\emptyset$

Postcondition. Converts the integer d into the string s.

function integer: d `StrLength` (string: s)

Precondition. $\emptyset$

Postcondition. d is the length of the string s.

function string: s' `StrSetChar` (string: s, integer: d, char: c)

Precondition. $0 \leq d <$`StrLength`(s) and $c \neq$`'^0^'`.

Postcondition. $\forall i \in \{0 \ldots$`StrLength`$(s) - 1\} \setminus \{d\} : s'_i = s_i$ and $s'_d = c$.
on error: $s' = s$.

function integer: d `StrStoi` (string: s)

Precondition. 1) The number represented by s is a number between `INT_MIN` and `INT_MAX`. 2) The string must match this regular expression: `[\t ]*[+-]?[0..9]+` (see appendix F for more details about regular expressions). The string may start with spaces or tabular characters but must end with a number. A number may have a sign (+ or -) and must have at least one digit.

Postcondition. Converts the longest prefix of s satisfying the above regular expression to an integer d. Space and tabular characters at the beginning of s are omitted.
on error: If the value d is outside the integer range, then the result d is undefined. If the string s does not match the regular expression, then $d = 0$.

function string: s' `StrToLower` (string: s)

Precondition. $\emptyset$

Postcondition. All the characters $c \in \{A, \ldots, Z\}$ in the string s are replaced by their corresponding lowercase letters, the result is stored in s'. No other characters are changed.

function string: s' `StrToUpper` (string: s)

Precondition. $\emptyset$

Postcondition. All the characters $c \in \{a, \ldots, z\}$ in the string s are replaced by their corresponding upper case letters, the result is stored in s'. No other characters are changed.

function integer `StrCmp` (string: s_1, string: s_2)

Precondition. $\emptyset$

Postcondition. Returns 0 if $s_1 = s_2$, 1 if $s_1 > s_2$ and -1 otherwise.

function integer `StrCmpLU` (string: s_1, string: s_2)

Precondition. $\emptyset$

Postcondition. Returns 0 if $s'_1 = s'_2$, 1 if $s'_1 > s'_2$ and -1 otherwise, where $s'_i =$ `StrToUpper`(s_i) and $i \in \{1, 2\}$.

function integer `StrIsInteger` (string: s)

Precondition. $\emptyset$

Postcondition. Returns 1 if `CharIsDigit`(s_i)=1 $\forall 0 \leq i \leq$ `StrLength`$s-$ 1 and 0 otherwise.

See also `MakeChoice` and `MakeChoiceLU` in chapter 9 (pages 65).

7.3 Operations on Lists and Cursors

procedure `AddFirst` (list: $l1$, any: e)

Precondition. $\emptyset$

Postcondition. After evaluation of the expression e, the result is added to the list $l1$ at the first position. If the expression is a list, this list is shared by $l1$.

procedure `AddLast` (list: $l1$, any: e)

Precondition. $\emptyset$

Postcondition. After evaluation of the expression e, the result is added to the list $l1$ at the last position. If the expression is a list, this list is shared by $l1$.

function any: r `GetFirst` (list: l)

Precondition. l is a non-empty list

Postcondition. r is the first element of the list l. Of course, if the first element of a the list is a list, then the result is not a copy of it but shares it.
on error: the program is halted.

function any: r `GetLast` (list: l)

Precondition. l is a non-empty list

Postcondition. r is the last element of the list l. Of course, if the last element of a the list is a list, then the result is not a copy of it but shares it.
on error: the program is halted.

function integer : n `Length` (list: l)

Precondition. $\emptyset$

Postcondition. n is the number of elements found in the list l.

function cursor: c `member` (list: l, any : m)

Precondition. $\emptyset$

Postcondition. if the element m occurs in the list l, then the cursor c points to this element. If the elements occurs more than once, then c points to the first occurrence. Elements that have a type different of the element m are omitted. If m does not belong to the list then the cursor c is void.

7.4 Operations on Files

function file `OpenFile` (string: FileName, integer: Mode)

Precondition. *FileName* is the name of a file. *Mode* is an integer constant among: `_W` for the write mode and `_R` for the read mode and `_A` for the append mode.

Postcondition. Depending on the value of *Mode*:

`_W` : If the file *FileName* exists then it is destroyed and the result is a handle to a new file opened for writing only. If *FileName* is not a valid name, the result is `void` and the error register is set to `ERR_FILE_OPEN`.

`_R` : If the file *FileName* does not exist, the result is the value `void` and the error register is set to `ERR_FILE_OPEN`. Otherwise the result is a handle to the file opened for reading. The *current position* is either the first character of the file or the end of file if the file is empty.

`_A` : If the file *Filename* does exist then the function returns an handle to this file opened for writing at the end-of-file. Otherwise, the file is created and the function behaves like the mode was `_W`.

procedure `CloseFile` (file: f)

Precondition. *f* denotes an handle to a file opened with the instruction `OpenFile`.

Postcondition. The file is closed, and the value of *f* is undefined.
on error: The error register is set to `ERR_FILE_CLOSE`.

procedure `printf` (file: f, any: value)

Precondition. *f* denotes a file opened for writing and *value* is any expression among types `string`, `char`, `integer`, `list`.

Postcondition. *value* is written on the file denoted by *f*. If the type of *value* is `list` then all the values found in this list are written on the file surrounded by the string constants `LEFT`,`RIGHT` and separated by the string constant `COMMA`[1] (*cfr.* 7.4 for more details about these constants). Values not belonging to types `string`, `char`, `integer`, `list` are skipped.
on error: The behavior is undefined.

Let us remark that very depth and recursive lists perturb this procedure. The procedure `print` only accepts the second argument of `printf` and write the value on the console.

[1] These constants are internal and are not visible.

function any `readf` (file: f, integer: t)

Precondition. *f* denotes a file opened for reading and *t* is an integer constant denoting the type of value to be read in the file. Following constants can be used: `_integer`, `_char`, `_string`.

Postcondition. Upon the value of *t*, the instruction will behave like this:

`_integer :` The longest sequence of decimal digits optionally preceded by - or + is read from the *current position*. At the end, the *current position* is either the first character after the sequence or the end of file. If *current position* is either the end of file or is not indicating a number, then 0 is returned. If the sequence denotes a number outside the range `[INT_MIN...INT_MAX]` then the instruction returns a random integer.

`_string :` The longest sequence of characters before either the end of file or the first character '\n' or the `MAX_BUFFER`*nth* character after the *current position*. If the *current position* is the end of file or is indicating the end of line character, then the empty string is returned. The *current position* becomes either the end of file or the first character after the sequence.

`_char :` The character under the *current position* is returned. If the end of file is reached, the ASCII code 0 is returned.

The function `read` behaves like `readf` except that characters are read from the console.

function integer `eof` (file: f)

Precondition. The file *f* is opened.

Postcondition. `eof` returns 1 if the end of file is reached and 0 otherwise.

function integer `neof` (file: f)

Precondition. The file *f* is opened.

Postcondition. `neof` returns 0 if the end of file is reached and 1 otherwise.

For files opened for writing, the function always returns 0. C programmers will note that the function `neof` is quite different of the function `feof` in this language.

Some other instructions are discussed here although they have no concern with the type `file`.

procedure `rename` (string: OldName, string: NewName)

Precondition. *OldName* is the name of an existing file. *NewName* is a file name that does not yet exist. Both expressions must denote files on a same physical device.

Postcondition. The file *OldName* is renamed *NewName*. If paths are different, then this instruction will move the file. On errors, the error register is set to `ERR_ERROR`.

procedure `delete` (string: filename)

Precondition. *filename* is the name of an existing file.

Postcondition. The file *filename* is deleted. On errors, the error register is set to one of the following values: `ERR_PERMISSION_DENIED`, `ERR_PATH_NOT_FOUND`.

function integer `ExistFile` (string: filename)

Precondition. *filename* is a valid file name for DOS. The file may not exist.

Postcondition. The function returns 1 if the file exists. Otherwise, error codes `ERR_PERMISSION_DENIED` and `ERR_PATH_NOT_FOUND` can be returned. The error register is not modified.

procedure `SetPrintList` (string: left, string: right, string: comma)

Precondition. *left*, *right* and *comma* are strings with no more than `MAX_DELIM` characters. Strings can be empty.

Postcondition. Strings *left*, *right*, lexicalcomma are put into constants `LEFT`, `RIGHT` and `COMMA`.

The next example illustrates the use of the previous instructions.

Example:

```
file:  f;
begin
    f:=OpenFile("c:\\tmp\\foo.txt",_W);
    SetPrintList("(",")",",");
    printf(f,[1,[1,2,3],4,'\n']);
    SetPrintList("\/*","*/\n","\n");
    printf(f,["line comment 1","line comment 2","line comment 3"]);
    CloseFile(f);
end
```

The program will print the next characters in the file `foo.txt`:

```
(1,(1,2,3),4)
/*line comment 1
line comment 2
line comment 3*/
```

□

7.5 Interface Operations

The following operations are illustrated with screen snapshots. Although the manual is written in English, my operating system has a French configuration, and therefore dialog boxes are a mix of French and English texts. French texts are system dependent messages and English are user's parameters defined below.

function string `DialogBox` (string : t, string :m, integer: s, string: d)

Precondition. $\emptyset$

Postcondition. Create a dialog box shown in figure 7.1 (page 50) from the argument interpreted as:

t**:** the title (`<TITLE>` in the figure)

m**:** the message (`<MESSAGE>` in the figure)

s**:** the maximum length of the input area in characters

d**:** the default string displayed in the input area (`<DEFAULT VALUE>` in the figure)

If the user chooses the `CANCEL` button, the result is the empty string and the error register is set to `ERR_CANCEL`.

function string `BrowsePrint` (string : t, string :m, string: e)

Precondition. $\emptyset$

Postcondition. Create a dialog box shown in figure 7.2 (page 52) from the argument interpreted as:

t**:** the title (`<TITLE>` in the figure)

m**:** the message (`<MESSAGE>` in the figure)

e**:** suggested extensions. This string is formatted as a list of pairs like this: `"name|ext|name|ext|name|ext"` where `name` is the associated name of one extension ("`text file`" for instance) and `ext` is its extension ("`*.v2`" for instance).

If the user chooses the `CANCEL` button, the result is the empty string and the error register is set to `ERR_CANCEL`. Otherwise, the result is name of the selected file (with its path). The user may either choose an existing file or type a new name.

function string `BrowseRead` (string : t, string :m, string: e)

Precondition. $\emptyset$

Postcondition. Create a dialog box shown in figure 7.2 (page 52) from the argument interpreted as:

t**:** the title (`<TITLE>` in the figure)

m**:** the message (`<MESSAGE>` in the figure)

e**:** suggested extensions. This string is formatted as a list of pairs like this: `"name|ext|name|ext|name|ext"` where `name` is the associated name of one extension ("`text file`" for instance) and `ext` is its extension ("`*.v2`" for instance).

If the user chooses the `CANCEL` button, the result is the empty string and the error register is set to `ERR_CANCEL`. Otherwise, the result is name of the selected file (with its path). Although file names are greyed, the user can type a new file name.


<TITLE>
<MESSAGE>

procedure `MessageBox` (string : t, string :m)

Precondition. $\emptyset$

Postcondition. Create a messge box displayed as in the figure 7.3 (page 53). Arguments are interpreted as:

t**:** the title (`<TITLE>` in the figure)

m**:** the message (`<MESSAGE>` in the figure)

function integer: r `Choice` (string : t, list :L, integer: s, integer: m)

Precondition. $\emptyset$

Postcondition. Create a dialog box that let the user to choose an item among several items.

t**:** the title (`<TITLE>` in the figure)

L**:** the list that will denote the possible items. The elements of L which do not match the `string` type will be omitted.

s**:** the listbox will be sorted if $s \neq 0$.

m**:** whenever the user will click on the OK button, an item must be selected

If the user chooses the `CANCEL` button, the result is -2. If the choice is not mandatory (m =`FALSE`) and if no item is selected, then the result is -1. Otherwise, the result is the index of the string in the list (the first index is 0). (See fig. 7.4).

7.6 Time Operations

function integer `GetDay` ( )

Precondition. $\emptyset$

Postcondition. returns the current day (1-31).

function integer `GetHour` ( )

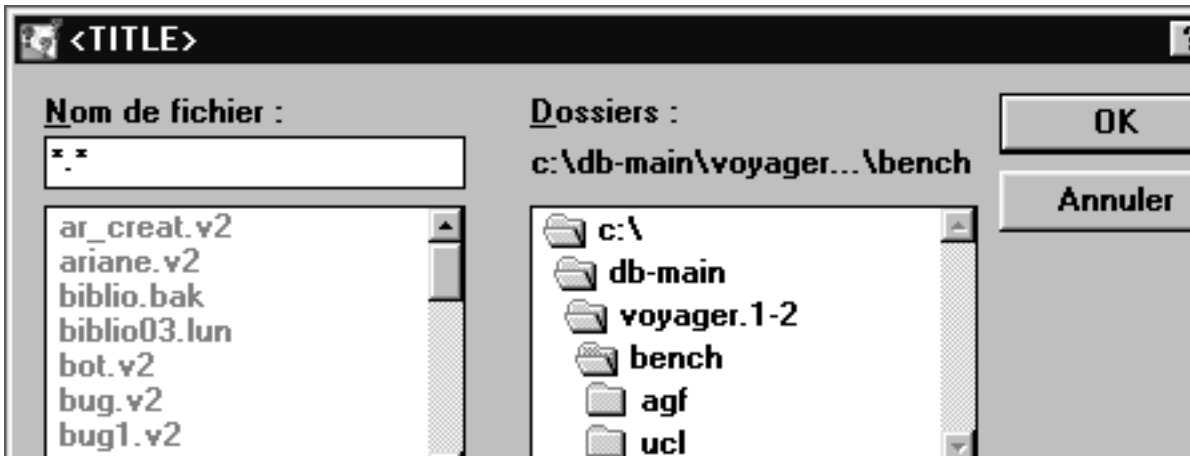
Precondition. $\emptyset$

Postcondition. returns the hour (0-23).

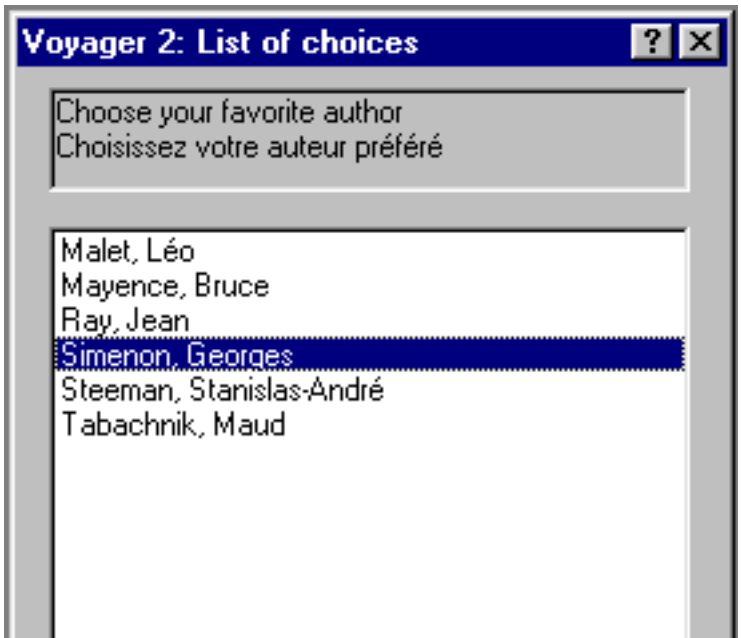
function integer `GetMin` ( )

Precondition. $\emptyset$

Postcondition. returns the minute (0-59).


<TITLE>
Nom de fichier :
.
Dossiers :
c:\db-main\voyager...\bench
OK
Annuler
ar_creat.v2
ariane.v2
biblio.bak
biblio03.lun
bot.v2
bug.v2
bug1.v2
c:\
db-main
voyager.1-2
bench
agf
ucl


<Title>


Voyager 2: List of choices
Choose your favorite author
Choisissez votre auteur préféré
Malet, Léo
Mayence, Bruce
Ray, Jean
Simenon, Georges
Steeman, Stanislas-André
Tabachnik, Maud

function integer `GetMonth` ( )

Precondition. $\emptyset$

Postcondition. returns the month (1-12).

function integer `GetSec` ( )

Precondition. $\emptyset$

Postcondition. returns the second (0-59).

function integer `GetWeekDay` ( )

Precondition. $\emptyset$

Postcondition. returns the day in the week (1-7).

function integer `GetYear` ( )

Precondition. $\emptyset$

Postcondition. returns the year.

function integer `GetYearDay` ( )

Precondition. $\emptyset$

Postcondition. returns the day in the year (1-365).

7.7 Flag Operations

Arrays of bits have been introduced in the repository definition to compact the project size. This new type needs ad-hoc functions to access each bit. An array of bit is stored into an integer and each bit is used as a boolean value. Integer having 32 bits, it is thus possible to store 32 boolean values in it.

function integer: r `GetFlag` (integer: d, integer: p)

Precondition. $\emptyset$

Postcondition. Returns the bit stored at position p in the integer d. The value p is a binary mask to access the bit.

function integer: r `SetFlag` (integer: d, integer: p, integer: v)

Precondition. $\emptyset$

Postcondition. Builds a new flag from d where the bit at position p has been set to v. All the other bits are let unchanged.

7.8 General Operations

procedure `BlackBoxP` (integer: c, ...)

Precondition. c denotes a unique code that must correspond to some defined operation. The "..." denotes the arguments of the procedure. This procedure is described in the help file of `DB-MAIN`.

Postcondition.

function any `BlackBoxF` (integer: c, ...)

Precondition. c denotes a unique code that must correspond to some defined operation. The "..." denotes the arguments of the procedure. This procedure is described in the help file of `DB-MAIN`.

Postcondition.

The `posx`, `posy`, and `color` attributes are no more accessible. The programmer must now use ad-hoc accessors named `GetPosX`, `GetPosY`, and `GetColor`. They accept two arguments: a `user_view` and a `generic_object`. They return a value (an integer) that corresponds to the expected attributed with respect to the view (i.e., a window). Procedures `UpdatePosX`, `UpdatePosY`, and `UpdateColor` accept a third argument (an integer) and udpates the ad-hoc attribute in the user view.

function integer `GetOID` (generic_object: g)

Precondition. $g \neq \emptyset$

Postcondition. Return the technical identifier of the generic object. This value is unique and stable.

procedure `call` (string: s)

Precondition. s denotes a windows application with optional arguments.

Postcondition. The program s is executed and the `Voyager 2` program continues its execution. **on error:** The error register is set to the constant `ERR_CALL`.

procedure `ClearScreen` ( )

Precondition. $\emptyset$

Postcondition. The screen is cleared.

function any `GetCurrentObject` ( )

Precondition. $\emptyset$

Postcondition. Returns the reference of the object that is selected. If no object is currently selected, then the function returns `Void(GENERIC_OBJECT)`.

function schema `GetCurrentSchema` ( )

Precondition. ∅

Postcondition. Returns the reference of the current schema. If no schema is selected, the function returns the value `void`.

function integer: e `GetError` ( )

Precondition. ∅

Postcondition. e is the value found in the error register. The call puts the value 0 in the error register.

function string: s `GetOxoPath` ( )

Precondition. ∅

Postcondition. returns a string with the path of the "oxo" file that contains the current program.

function integer: r `GetType` (any: v)

Precondition. ∅

Postcondition. this function returns the value denoting the accurate type of the value passed as argument. For instance, if the argument is a variable defined as `ent_rel_type` then the possible results are `ENTITY_TYPE` and `REL_TYPE`. This function is mainly used to know how to process the values coming from lists when lists are heterogeneous.

function integer: r `IsNoVoid` (any: v)

Precondition. ∅

Postcondition. returns `not IsVoid(v)`. See 7.8 for more details.

function integer: r `IsVoid` (any: v)

Precondition. ∅

Postcondition. returns `TRUE` if the argument is null. If the type of the argument is `list` or `string` the result is always `FALSE`. For integers and characters, this predicate is true if the value is the integer 0 or the character `'^0^'`.

function any: o `Void` (integer: t)

Precondition. The integer constant t denotes a valid type of the language, except the types: `list` and `string`.

Postcondition. o is the special value `void` of the type denoted by t.
on error: The program halts.

Chapter 8

Functions and Procedures

8.1 Definition

Functions and procedures are abstractions of program slices. We make no distinction between both terms except if it is explicitly mentioned (in this case, the term is underlined like that: function). The scope of a function is the whole program. Each function is identified by its name. The definitions may occur anywhere between the last global variable definition and the main program. Functions may have local variables (their scope is restricted to the body of the function) and return a value (the result) of any type (not the procedures!). The syntax of a function definition is formally defined in A.

For functions, the flow of the executed instructions must pass through a `return` instruction before reaching the end of the function. If this condition is not respected, the execution will probably be aborted with a strange message! Functions returning no values have no sense in `Voyager 2`[1].

In the body of a function, `return` instructions may occur anywhere in the body and must be followed by an expression whose the type is exactly the same has the one specified in the definition. When this instruction is reached, all the local variables disappear from the environment, the memory is cleaned and the expression is returned as the result of the function.

In the body of a procedure, the `return` instruction as the same sense as before except that no expression may follow it[2]. If the flow of the executed instructions reaches the end of the body, then the execution of the procedure is completed and all the local variables are removed from the environment.

When a function is completed, the flow is directed to the next instruction following the function call.

All the arguments are passed by value except for lists. This means, that for each call, the parameter is in fact a copy of the argument. But let us note two remarks:

1. lists are passed by *reference*. This means that all the operations performed on a list parameter are also performed on the argument.

[1] This is another difference with the C language.

[2] If an expression follows the `return` statement, then this expression is not evaluated. Its presence is not a syntax error but has no influence on the program

2. when arguments are reference to objects, although this argument is passed by value, the behavior of the program is the same as if the value was passed by reference. The reason is simple:

   - the value is persistent and is stored in a repository.
   - if a variable is a reference to one object, then this variable is like a pointer and then is quite the same thing than passing the object by reference[1].

The remark about list arguments also holds for lists returned by functions. Let us suppose that l is a local list variable, and its value before the call to the `return` instruction is `[1,2,3]`. Then this list is still valid after the call although we said that all the local variables were destroyed when a function exits. The reason is very simple: lists are managed by a *garbage collector* and this one sees that the list is both used by a local variable and by the program calling the function. Hence, the garbage collector does not destroy the list.

Finally, there is no implicit type casting argument expressions to the type of the arguments specified in the signature of the function. This means that this job must be done by the programmer. For instance, if a function expects a *data_object* as first argument, it is forbidden to pass expressions of another type (even *entity_type*) that is a subtype.

The following program illustrates the use of functions and procedures to print lists of factorials:

```
function integer fact(integer:  n)
    integer:  i, f;
{   f:=1;
    for i in [1..n] do {
       f:=f*i;
    };
    return f;
}

procedure PrintFact(integer:  i, integer:  j)
    integer:  z;
    list:  l;
{   for z in [i..j] do {
       l:=l++[fact(z)];
    };
    print(l);
}

begin
    SetPrintList("","",",");
    PrintFact(2,5);
end
```

The `export` and `explain` clauses that appear in the syntax are outside the scope of this chapter and will respectively be described in 16.2 and 16.5.

[1] C programmers certainly know this technic.

8.2 Recursiveness

We said before that the scope of a function was the whole program. It is still true but can we call a function from inside its body? The answer is `yes!` This principle is called *recursiveness* and is very useful in practice. This is specially true in `Voyager 2` since the stack used by `Voyager 2` is much bigger than the one used by classical languages like C and Pascal (for MS-DOS/WIN3.1 only). For instance the factorial function could be shortened in this way:

```
function integer fact(integer:  n){
    if n=0
    then { return 1; }
    else { return n*fact(n-1); };
}
```

In the same way, the `PrintFact` procedure could be rewritten like that:

```
procedure PrintFact(integer:  i, integer:  j){
    if i<j
    then {
       print(fact(i));
       print(',');
       PrintFact(i+1,j);
    } else {
       print(fact(i));
    };
}
```

Chapter 9

Lexical Analyzer

Because strings are passed by value to functions and procedures in `Voyager 2` and also because characters are read once a time from files, so far, it was not easy to write efficient lexical analyzers in `Voyager 2`. For these reasons, some specificic functions were added to do this job.

All these functions use the same *input stream* that is initialized by the function `SetParser`. The input stream may be either a file or a string. Because a stream is a little bit more sophisticated than a normal file (`OpenFile`), usual functions for files can not be used with this *input stream*.

procedure `SetParser` (τ: sf)

Precondition. τ is either the `string` type or the `file` type. This instruction specifies from which stream the functions from the lexical library will read the input. If the argument is a string, then all the lexical functions will read characters from a "virtual" file initialized with the argument. Otherwise, if it is a file, characters are read from the file itself.

Postcondition. The input stream is initialized.

function string : r `GetTokenWhile` (string : s)

Precondition. The input stream is initialized.The argument must be a literal string – the value of s must be known at the compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension or in expansion. The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded like that: for each occurence of "α-β" where α, β denote any character, this substring is replaced in the pattern by the set of characters $\gamma : \alpha \leq \gamma \leq \beta$. Thus the following pattern : "`-a-d0-4`" is equivalent to the string: "`-abcd01234`".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. The result of this function is the string $(\alpha_i)_1^m$ where $0 \leq m \leq n$ and $\forall i \in 1..m : \alpha_i \in \mathcal{P}$ and if $m < n$ then $\alpha_{m+1} \notin \mathcal{P}$ where $\mathcal{P}$ is the pattern. After the call, the input stream is replaced by $(\alpha_i)_{m+1}^n$.

function string : r `GetTokenUntil` (string : s)

Precondition. The input stream is initialized. The argument must be a literal string – the value of s must be known at the compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension or in expansion. The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded like that: for each occurence of "α-β" where α, β denote any character, this substring is replaced in the pattern by the set of characters $\gamma : \alpha \leq \gamma \leq \beta$. Thus the following pattern : "`-a-d0-4`" is equivalent to the string: "`-abcd01234`".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. The result of this function is the string $(\alpha_i)_1^m$ where $0 \leq m \leq n$ and $\forall i \in 1..m : \alpha_i \notin \mathcal{P}$ and if $m < n$ then $\alpha_{m+1} \in \mathcal{P}$ where $\mathcal{P}$ is the pattern. After the call, the input stream is replaced by $(\alpha_i)_{m+1}^n$.

procedure `SkipWhile` (string : s)

Precondition. The input stream is initialized. The argument must be a literal string – the value of s must be known at the compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension or in expansion. The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded like that: for each occurence of "α-β" where α, β denote any character, this substring is replaced in the pattern by the set of characters $\gamma : \alpha \leq \gamma \leq \beta$. Thus the following pattern : "`-a-d0-4`" is equivalent to the string: "`-abcd01234`".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. After the call, the input stream is replaced by $(\alpha_i)_{m+1}^n$ where m is defined as $0 \leq m \leq n$ and $\forall i \in 1..m : \alpha_i \in \mathcal{P}$ and if $m < n$ then $\alpha_{m+1} \notin \mathcal{P}$ where $\mathcal{P}$ is the pattern.

procedure `SkipUntil` (string : s)

Precondition. The argument must be a literal string – the value of s must be known at the compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension or in expansion. The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded like that: for each occurence of "α-β" where α, β denote any character, this substring is replaced in the pattern by the set of characters $\gamma : \alpha \leq \gamma\beta$. Thus the following pattern : "`-a-d0-4`" is equivalent to the string: "`-abcd01234`".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. After the call, the input stream is

replaced by $(\alpha_i)_{m+1}^n$ where m is defined as $0 \leq m \leq n$ and $\forall i \in 1..m : \alpha_i \notin \mathcal{P}$ and if $m < n$ then $\alpha_{m+1} \in \mathcal{P}$ where $\mathcal{P}$ is the pattern.

procedure `UngetToken` (string : s)

Precondition. The input stream is initialized. Let $(\alpha_i)_1^n$ be the input stream. α_1 is the first character. Let $(\sigma_j)_1^m$ be the sequence of letters in s.

Postcondition. The input stream is replaced by $(\sigma_j)_1^m \circ (\alpha_i)_1^n$ where $\circ$ is the "append" operator for lists.

function char : c `GetChar` ( )

Precondition. The input stream is initialized. There is at least one character in the input stream.

Postcondition. The first character of the input stream is removed and returned.

function integer `seof` ( )

Precondition. The input stream is initialized.

Postcondition. Let n be the value returned by this function. Then $n = 0$ if there is one character in the input stream and $n \neq 0$ otherwise.

function integer `nseof` ( )

Precondition. The input stream is initialized.

Postcondition. Let n be the value returned by this function. Then $n \neq 0$ if there is one character in the input stream and 0 otherwise.

The following functions are not really in the lexical library. They should be defined in the section 7.2. But they are often used with lexical functions.

function integer: d `MakeChoice` (string: s, list: l)

Precondition. l is a literal list of strings —This value must be known at the compilation time. We note $(\sigma_i)_1^n$ the list l where σ_1 is the first element. All the values should be distinct.

Postcondition. if $\exists i \in 1..n : \ \sigma_i = s$ then $d = i$ otherwise $d = 0$. The complexity of the this function is $\Theta(\log_2 n)$. **on error:** If one value occurs several times in the list l, then the result is random. The compiler prints a warning message.

function integer: d `MakeChoiceLU` (string: s, list: l)

Precondition. l is a literal list of strings —This value must be known at the compilation time. We note $(\sigma_i)_1^n$ the list l where

σ_1 is the first element. All the values should be distinct. labelimakechoicelu

Postcondition. if $\exists i \in 1..n : \texttt{StrCmpLU}(\sigma_i, s) = 0$ then $d = i$ otherwise $d = 0$. This comparaison is case **in**sensitive. The complexity of the this function is $\Theta(\log_2 n)$. **on error:** If one value occurs several times (*case insensitive*) in the list l, then the result is random. The compiler prints a warning message.

Part II

The Repository

Chapter 10

Repository Definition

This chapter is devoted to a global presentation of the repository of DB-MAIN. The repository is composed of objects. Each one is formally defined (*cfr.* chapter 11) and all the instances of an object type must respect its definition. Two kinds of relations may exist between object types: *is-a* and *link* relations.

The *is-a* relation denotes the generalization concept. When an object A derives from an object B, we say that A (resp. B) is a specialization (resp. generalization) of B (resp. A). This simply means that all the properties of the object A also hold for the object B.

A link relation denotes a one-to-many relation between objects. If such a relation does exist between objects A and B as depicted in the following picture this means that to each instance of object A corresponds a collection[1]

To each side of a link r, one says that the object plays a role named r or $@r$ depending on the valuation of the cardinality of the role. In our example, object B plays the role $@r$ and A plays the role r. The cardinality of a role may be 0-1, 1-1 or 0-∞. In the first case, an instance may play at most one role, in the next case, each instance must play exactly one role and at least, in the last case, each instance may have several roles. ⌜***5.0***⌝

The repository of the version 5.0 is now too large to be presented in one piece. For this reason, the repository definition has been "exploded" in five views depending on the various ontologies the repository can model. The first view (cfr. Fig. 10.1) is a macro-view. This corresponds mainly to the objects the software engineer can observe in the project window. The "data-schema" view (cfr. Fig. 10.2) corresponds to the former definition, that is, the representation of the entity-relationship schemas. The third view (cfr. Fig. 10.3)is new and represents the process schema (statements, functions, expressions, ...). The fourth view (cfr. Fig. 10.4)denotes the persistent data that underlie the graphical representation of the schema (data and process). Finally, the last view (cfr. Fig. 10.5)is just an overview of all the objects which inherit from the `generic_object`. Of course, all those views are intimately linked together and objects can occur in several views. ⌞***5.0***⌟

Although each object type will be fully described and defined in chapter 11, we will give here a general overview of this schema. Let us remember that the aim of this schema definition is to store the definition of any extended

[1] this collection may be empty depending on the value of the cardinality.

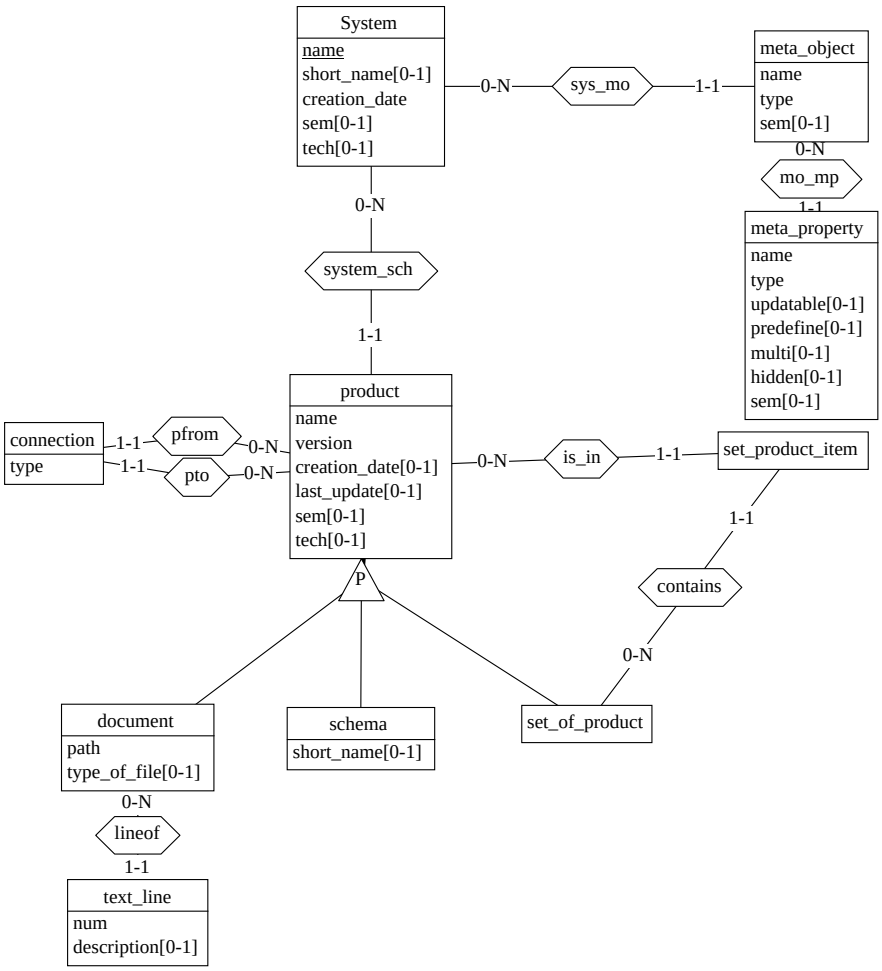


Figure 10.1: The “macro” view.

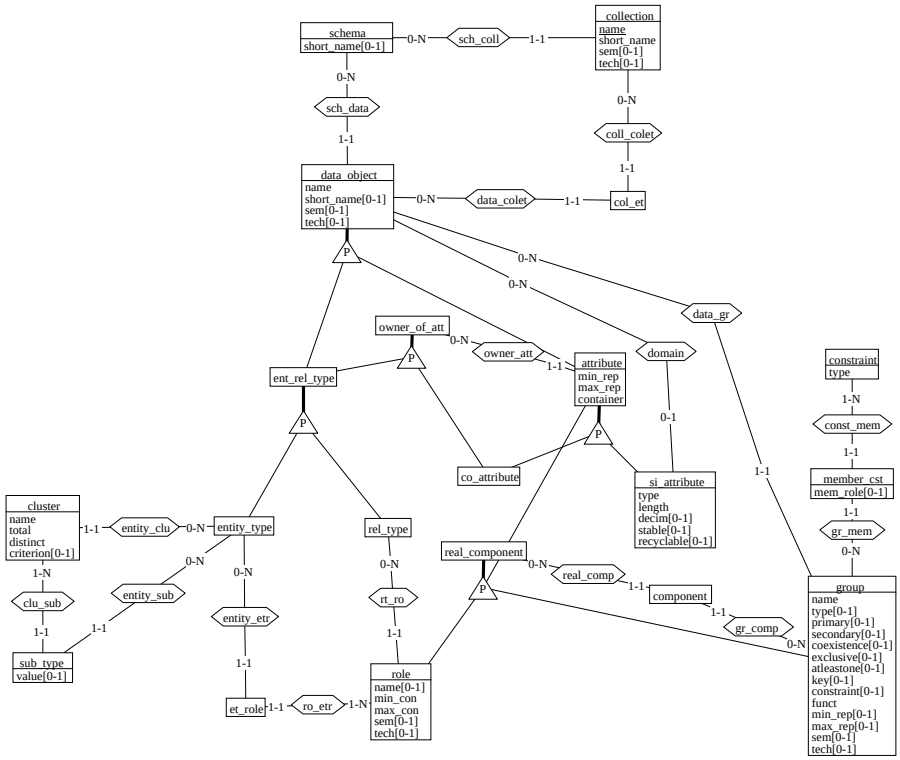


Figure 10.2: The “data” view.

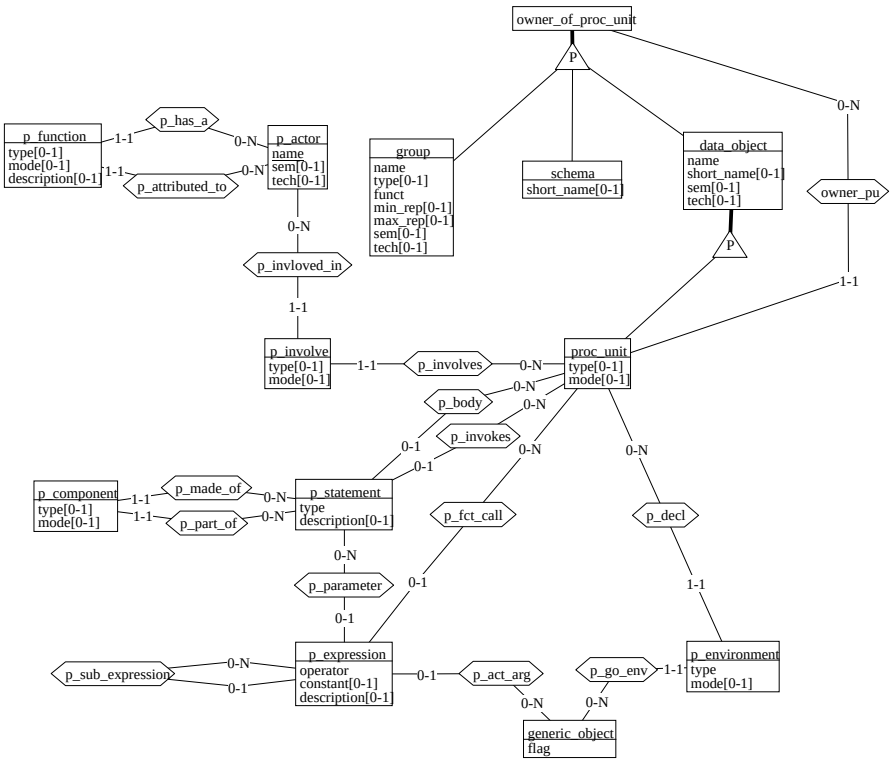


Figure 10.3: The “process” view.

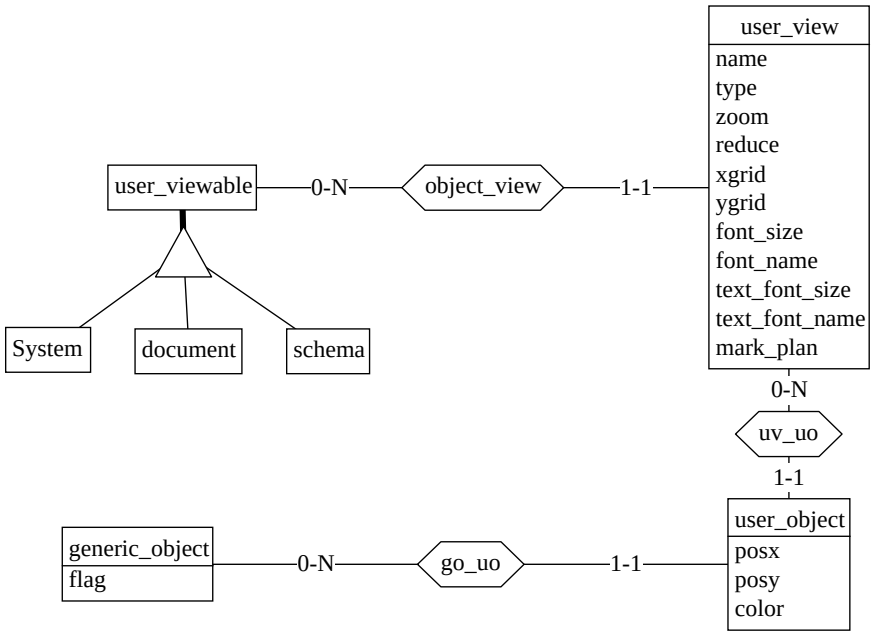


Figure 10.4: The “graph” view.

entity relationship schema. For this reason, one will need to represent concepts like *entity types*, *attributes*, *relationships*, ... To each concept will correspond an object type. For instance, the `entity_type` object type corresponds to the *entity type* concept and `co_attribute` to a *compound attribute*. However, some object types do not correspond to pertinent concepts. It is the case of the `component` object type. Its presence is due to technical reasons[1].

Our tour begins with the `system` object type. There is only one instance/object of this type. This object will denote the whole project. A system can be composed of schemas (`schema`) and documents (`document`). Because these object types share common propreties, we decided to define a more general object type: the `product`. Thus a product can be either a schema or a document. Products can be connected together via connections (`connection`). A schema denotes of course an ER schema and will contain entity types, relationship types, ... The three obvious concepts from the ER theory: the entity types, the relationship types and the attributes are represented by the `entity_type`, `rel_type` and `attribute` object types. There are three possible specializations of an attribute: it can be a simple attribute, or a compound attribute or a domain[2] attribute. To each one corresponds one specific object type: `si_attribute`, `co_attribute` and `do_attribute`. When a simple attribute denotes an object attribute (that is an attribute wich references an entity type), one links this attribute to its "type" with the `domain` link. Entity types, relationship types and compound attributes may all be composed of attributes. For this reason, the `entity_type`, `rel_type` and `co_attribute` inherit from the `owner_of_att` object type. This object type may own attributes (by using the link `owner_att`). The `entity_type` and `rel_type` object types inherit from the `ent_rel_type` object type that inherits itself from the `data_object` object type. We said above that a schema was composed of entity types, relationship types and attributes. All the object types that represent these concepts inherit from the `data_object` object type. Thus a schema will just contain objects from the `data_object` object type (the link `sch_data`).

Besides all these concepts may own properties that are represented by groups in the `DB-MAIN` tool. For this reason, a link (`data_gr` does exist between `data_object` and the new object type: `group`. A group is just a set of properties/concepts like attributes, roles, or even groups. A group alone has absolutely no semantics (this is explained afterwards). Let us consider the concept c (it is for example the attribute: "customer-name"). This concept may belong to several groups g_1, g_2, ..., g_n but we also said that one group may contain several items. Because one can not represent this *many-to-many* relationship type in our schema with links, we have introduced the `component` object type. Thus a group may be connected to several components, and one item/concept/real_component may be connected to several components. A real_component is just one instance of the `real_component` object type that is a generalization of all the possible concepts one can find in a group: attributes, roles and groups. We said that groups had no semantics. This is only true if no constraints are attached to it. A constraint can be a *referential integrity rule* for instance. Each constraint is represented by one instance/object of the `constraint` object type. To represent the *many-to-many* relationship type between `constraint` and `group`, we

[1] Our repository technology can not represent *many-to-many relationship types*.

[2] This concept is not yet managed by `DB-MAIN`.

have defined the `member_cst` object type *wrt* the same principle than the one explained above.

One relationship type (ie: one object of `rel_type`) may have several roles, each one being attached to at least one entity-type. The role concept is represented by the `role` object type. And the `et_role` object type is defined to implement the *many-to-many* relationship type between `entity_type` and `role`.

To each entity type may correspond several clusters (`cluster`). Each one describes a decomposition of the original entity type (the supertype) into subtypes (`sub_type`). To each subtype corresponds exactly one entity type. This decomposition is also named *is-a/inheritance* relation. Because multipe-inheritance is also representable in `DB-MAIN`, one entity type may own several subtypes via the `entity_sub` link.

Schemas may have collections (`collection`). Collections have no semantics but are often used as representation of concepts like *files*, *clusters*, *db-space*[1], *storage areas*[2], *areas*[3], ... Collections can thus own entity types and one entity type may belong to several collections (once again, the `coll_et` object type is just defined to represent a *many-to-many* relationship type between `collection` and `entity_type`).

A special object type has been created in the schema definition: the `generic_object` object type. This object type is a supertype of all the object types cited above.

[1] DB2, SQL92
[2] RDB.
[3] CODASYL.

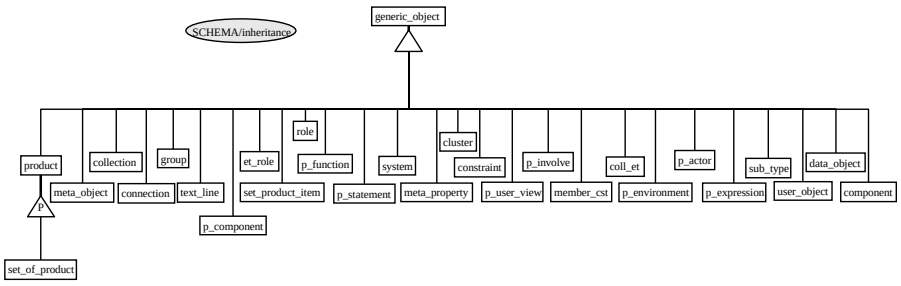


Figure 10.5: The “inheritance” view.

Chapter 11

Objects Definition

The repository of DB-MAIN is composed of object types related together by *is-a* relations and one-to-many relationships (network technology with inheritance). This chapter introduces definitions for each object of the repository and describes how to create them.

Each time a new object is defined, one describes the way to create such an object as well as the component of the object. Here follows the conventions we use.

For each object, a table will describe the formal components of the object like fields, roles and inheritance with all needed information about each one. Then, the semantics of the object is explained and finally one describes the way to create new objects.

The table looks like that example:

employee		
$\xrightarrow{\text{isa}}$ person		
name	string [L_NAME]	1–1
own	0–∞	cars

The first line contains the object name. Then come all the properties of the object: attributes and roles.

Each field is described by its name, its type and its minimum-maximum repetitivity. When an attribute is optional, the default value is also mentioned between parentheses. When the type is a string, the maximum length of the string is put after the type in a smaller size[1]. The size is indicated by the use of a predefined integer constant, we recommend to use the name of the constants rather than their value for obvious reasons of maintenance.

Each role is described by its name[2], the minimum-maximum cardinality and finally the object present at the other side of the link.

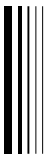
Each time a new object is created, its environment must be defined. The environment of an object is the collection of all the informations required to create it: fields values, roles, positions... Among these informations, some are

[1] Although strings size is unlimited in Voyager 2, the need to store this information in a persistent way on disk requires to limit the size of each string to a reasonable length.

[2] Let us note that roles name are sometimes preceded by the symbol @ to indicate which direction of the link is used.

mandatory and others not. When a role is mandatory, its name is printed in bold type in the table.

The creation of objects is made by calling the function `create` with in arguments the components of the environment. Such a call looks like that:



```
et:=create(ENTITY_TYPE,
     name:"employee",
     short_name:"emp",
     sem:"male/female",
     @SCH_DATA:GetCurrentSchema());
```

where the first argument is a predefined integer constant denoting the type of the object to create. The next arguments are items composed of two expressions: a constant denoting the component of the environment being specified and the second one is its value. All the mandatory components must be specified, and the order is without importance. If a component is specified twice or more, `create` will choose one of them randomly. In an item, when the left expression of the ':' operator is underlined, this one is required in the environment. Otherwise it is optional but subject to restrictions explained in the semantics description.

If a mandatory component (field, role, ...) is lacking, or if an integrity constraint is violated then the program is aborted! There is no sense for `Voyager 2` to treat these errors since they should be prevented by the programmer. When a program is aborted for such reasons, a possible explanation is printed on the console, but sometimes the message can be confusing. It is the case for instance, when unicity/integrity constraints are violated[1], in such cases, the program is aborted and a message about the memory is displayed[2]. But in all cases, you will know which instruction is causing the error.

We said that the repository was implemented with *one-to-many relationships*. This kind of relationship is called a *link*. This terminology make easier the distinction between relationships used by the repository (*link*) and relationships defined by the repository (see 11.13). They always are two types of objects on each side of a link: the *father* and the *son*. Obviously, the father may have zero, one or more sons and sons may have zero or one father but no more. To each link corresponds one integer constant described in the table 2.5.

11.1 generic_object

generic_object		
flag	`integer`	0–1 (= 0)
p_act_arg	0–∞	`p_expression`
p_go_env	0–∞	`p_environment`
go_uo	0–∞	`user_object`

This object type has no other properties than being a supertype of all the other object types. This property if often used when programmers have to deal with objects for which they do not know the exact type. For instance: "What is the current selected object in the schema window?" (`GetCurrentObject()`)

[1] For instance: you are defining for the second time an entity-type with the same name.

[2] This is due to improve the efficiency of the system.

will return one object whose the type is unknown. The programmer can store this value in a variable of type `generic_object`.

The `flag` field is an array of bits that stores various information. The "mark information" is such an information. The programmer can use the functions `GetFlag` and `SetFlag`[1] to manage this array.

Let us note that this array can store information you are are not aware. Therefore, be careful to preserve all the bits whatever is the bit you are interested in. So far, five indexes have been defined: `MARK1`, `MARK2`, `MARK3`, `MARK4`, `MARK5`. The flag of index $\texttt{MARK}_i$ is true iff the corresponding object belongs to the blazing number i (see figure 11.1). The next example shows how using this flag:

```
schema: sch;
data_object: dta;
integer: the_flag;

begin
  sch:=GetCurrentSchema();
  for dta in DATA_OBJECT[dta]{@SCH_DATA:[sch] with GetType(dta)=ENTITY_TYPE} do {
    the_flag:=dta.flag;
    if GetFlag(the_flag,MARK2) then {
      dta.flag:=SetFlag(the_flag,MARK2,FALSE);
    } else {
      dta.flag:=SetFlag(the_flag,MARK2,TRUE);
    }
  }
end
```

This program marks unmarked entity types and unmarks marked entity types (wrt. the 2^{th} blazing). The last index is `SELECT` that is used to select/unselect an object.

11.2 user_object

user_object		
$\xrightarrow{\text{isa}}$ generic_object		
posx	`integer`	0–1 (= unknown)
posy	`integer`	0–1 (= unknown)
color	`integer`	0–1 (= unknown)

This object type denotes a position inside an Entity-Relationship schema. The `posx` and `posy` fields denote the graphical positions.

[1] *cfr.* page 55

11.3 system

system		
$\xrightarrow{\text{isa}}$ user_viewable		
name	`string` [L_NAME]	1–1
short_name	`string` [L_SNAME]	0–1 (= " ")
creation_date	`string` [L_DATE]	0–1(see †)
sem	`string`	0–1 (= " ")
tech	`string`	0–1 (= " ")
sys_sch	0–∞	`product`

An object of type `system` represents the whole project under development: schemas, processes, files (document), ... There is only one object of this type at the same time since `DB-MAIN` can not manage several projects. A system is composed of products.

To create a new `system`:

what:=create(`SYSTEM`, `name`:*string*, `short_name`:*string*, `creation_date`:*string*, `flag`:*integer*);

† When the `creation_date` field is not mentioned, then this field if automatically updated with the date of the computer.

11.4 product

product		
name	`string` [L_NAME]	1–1
version	`string` [L_VERSION]	1–1
creation_date	`string` [L_DATE]	0–1(see †)
last_update	`string` [L_DATE]	0–1(see †)
sem	`string`	0–1 (= " ")
tech	`string`	0–1 (= " ")
@system_sch	1–1	`system`
pfrom	0–∞	`connection`
pto	0–∞	`connection`

Objects of type `product` do not exist per se. They only have a sense as a generalization of schemas (*cfr.* section 11.5) and documents (*cfr.* section 11.8). Therefore it is impossible to create directly such an object. Each product is identified by its mandatory components.

† It is recommended not to use the `creation_date` and `last_update` fields since they are updated automatically. In case these fields are used, they must follow the following format : "*YYYYMMDD*", exactly four digits to represent the year, exactly two digits for the month and exactly two digits for the day.

11.5 schema

schema		
$\xrightarrow{\text{isa}}$ product $\xrightarrow{\text{isa}}$ user_viewable $\xrightarrow{\text{isa}}$ owner_of_proc_unit		
short_name	`string` [L_SNAME]	0–1 (= " ")
sch_coll	0–∞	`collection`
sch_data	0–∞	`data_object`

Objects of type `schema` are generalized entity-relationship schemas. Each schema belongs to only one system[1] and is identified by its name, its version and the system.

The following indexes in the `flag` field denote the following properties:

`RTSQUARE:` the shape of the rel-types has square corners.

`RTROUND:` the shape of the rel-types has round corners.

`RTSHADOW:` the rel-types are displayed with a shadow.

`ETSQUARE:` the shape of the entity types has square corners.

`ETROUND:` the shape of the entity types has round corners.

`ETSHADOW:` the entity types are displayed with a shadow.

⌜*6.0*⌝ To create a new `schema`:

```
what:=create(SCHEMA,
    name:string,
    short_name:string,
    version:string,
    creation_date:string,
    last_update:string,
    flag:integer,
    sem:string,
    tech:string,
    SCHEMA_TYPE:schema_type,
    @SYSTEM_SCH:system);
```

⌞*6.0*⌟ Note: the `SCHEMA_TYPE` field is to be used only in conjunction with the MDL language for defining methods [7].

11.6 set_of_product

set_of_product		
$\xrightarrow{\text{isa}}$ product $\xrightarrow{\text{isa}}$ owner_of_proc_unit		
contains	0–∞	`set_product_item`

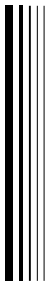
[1] There is only one `system` instance in the repository of `DB-MAIN`.

Objects of type `set_of_product` are denote sets of products. Every object belongs to only one system[1] and is identified by its name, its version and the system.

The following indexes in the `flag` field denote the following properties:

`HIDEPROD:` Will `DB-MAIN` must display the elements of the set?

To create a new `set_of_product`:



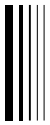
```
what:=create(SET_OF_PRODUCT,
    name:string,
    version:string,
    creation_date:string,
    last_update:string,
    flag:integer,
    sem:string,
    tech:string,
    @SYSTEM_SCH:system);
```

11.7 set_product_item

set_product_item		
$\xrightarrow{\text{isa}}$ generic_object		
@contains	1–1	`set_of_product`
@is_in	1–1	`product`

Each `set_product_item` object denotes a tuple (p, s) where p is an product and s is a set of products. Each tuple (p, s) means that the product p belongs to the set of products s.

To create a new `set_product_item`:



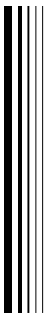
```
what:=create(SET_PRODUCT_ITEM,
    flag:integer,
    @CONTAINS:set_of_product,
    @IS_IN:product);
```

11.8 document

document		
$\xrightarrow{\text{isa}}$ product		
$\xrightarrow{\text{isa}}$ user_viewable		
path	`string`	1–1
type_of_file	`string`	0–1 (= `" "`)

Objects of type `document` denote any document like files, documentation, ... The `type_of_file` field is a string describing the type of file (*eg.* "text", "COBOL", "annual report 95", ...). The `creation_date` and `last_update` fields are optional (*cfr.* section 11.4 for more details about the default value). Documents are attached to one and only one schema.

To create a new `document`: ⌜6.0⌝



what:=create(DOCUMENT,
name:*string*,
version:*string*,
path:*string*,
creation_date:*string*,
last_update:*string*,
flag:*integer*,
type_of_file:*string*,
DOCUMENT_TYPE:*schema_type*,
@SYSTEM_SCH:*system*);

⌞6.0⌟ Note: the `DOCUMENT_TYPE` field is to be used only in conjunction with the MDL language for defining methods [7].

11.9 connection

connection		
$\xrightarrow{\text{isa}}$ generic_object		
type	`string` [L_ROLE]	0–1 (= " ")
@pfrom	1–1	`product`
@pto	1–1	`product`

Objects of type `connection` establish oriented links (vertices) between objects like schemas and documents (file, ...). If the `type` field is not mentioned, then the default value is " ". This field can have any value but someones have precise semantics and are described hereafter:

`CON_COPY:` The target product is a copy of the origin product.

`CON_DIC:` The target product is a "printed" copy (File/Print dictionary).

`CON_GEN:` The target product was generated from the origin product.

`CON_INTEG:` Undocumented feature (so far).

`CON_XTR:` The target product is the result of the extraction process.

To create a new object of type `connection`:

what:=create(CONNECTION,
type:*string*,
flag:*integer*,
@PFROM:*product*,
@PTO:*product*);

[1] There is only one `system` instance in the repository of `DB-MAIN`.

11.10 data_object

data_object		
$\xrightarrow{\text{isa}}$ user_object		
$\xrightarrow{\text{isa}}$ owner_of_proc_unit		
name	`string` [L_NAME]	1–1
short_name	`string` [L_SNAME]	0–1 (= " ")
sem	`string`	0–1 (= " ")
tech	`string`	0–1 (= " ")
@sch_data	1–1	`schema`
data_gr	0–∞	`group`
domain	0–∞	`si_attribute`

Objects of type `data_object` are generalizations of objects denoting entity types, relationship types and attributes. Therefore, it is impossible to create objects of this type but only specializations like entity types (section 11.12), relation types (section 11.13) and attributes (sections 11.15 and 11.16). Data objects are the main components of a schema.

11.11 ent_rel_type

ent_rel_type
$\xrightarrow{\text{isa}}$ data_object
$\xrightarrow{\text{isa}}$ owner_of_att

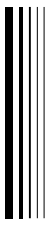
Objects of type `ent_rel_type` are generalizations of entity types (section 11.12) and relationship types (section 11.13). Therefore it is impossible to create such objects.

11.12 entity_type

entity_type		
$\xrightarrow{\text{isa}}$ ent_rel_type		
entity_etr	0–∞	`et_role`
entity_sub	0–∞	`sub_type`
entity_clu	0–∞	`cluster`

Each object of type `entity_type` denotes an entity type of a entity-relationship schema. Entity types can belong to collections, have attributes, play roles in relations, and be generalization/specialization of other entity types.

To create a new object of type `entity_type`:



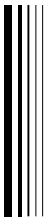
what:=create(`ENTITY_TYPE`,
`name`:*string*,
`short_name`:*string*,
`sem`:*string*,
`tech`:*string*,
`flag`:*integer*,
`@SCH_DATA`:*schema*);

11.13 rel_type

rel_type		
$\xrightarrow{\text{isa}}$ ent_rel_type		
rt_ro	0–∞	`role`

Objects of type `rel_type` denote relationships in ER schemas. Each relation is attached to exactly one schema and its name identifies it among all other relations. Rel-types may own attributes, groups and roles.

To create a new relation:



what:=create(REL_TYPE,
`name`:*string*,
`short_name`:*string*
`sem`:*string*,
`tech`:*string*,
`flag`:*integer*,
`@SCH_DATA`:*schema*);

11.14 attribute

attribute		
$\xrightarrow{\text{isa}}$ data_object $\xrightarrow{\text{isa}}$ real_component		
min_rep	`integer`	1–1
max_rep	`integer`	1–1
container	`char`	0–1 (= `SET_CONTAINER`)
@owner_att	1–1	`owner_of_att`

Objects of type `attribute` do not exist per se. The `attribute` is a supertype of types `si_attribute` and `co_attribute`. Therefore, there is no reason to create such objects. See sections 11.15 and 11.16 for more details.

All the fields of an attribute are shared with sub-types by the inheritance principle. The `min_rep` (resp. `max_rep`) is the minimum (resp. maximum) cardinality of the attribute. This value must be comprised between 0 and `N_CARD` which is equivalent to the infinite value[1]. The minimum cardinality must be less than the maximum cardinality. If one of the rules cited above is transgressed, then the program is aborted.

The `container` field denotes the kind of container used if the attribute is multivalued. The possible values of this field are:

`SET_CONTAINER`: It is the default value. It denotes a set as in mathematics.

`BAG_CONTAINER`: It is a set with possible duplicates. There is no order inside a set/bag.

`ARRAY_CONTAINER`: It is an array, each item can be referenced with an index. Arrays can contain several identical items.

`UNIQUE_ARRAY_CONTAINER`: It is an array but each element occurs only once.

[1] Only the maximum cardinality may take the infinite value.

`LIST_CONTAINER`: A list is a collection of items (duplicates are allowed). This collection is ordered.

`UNIQUE_LIST_CONTAINER`: A list with no duplicates.

Let us remark an interesting property of the environment: although the `@sch_data` was required in the `data_object` environment, it is no more the case here. The reason is very simple. This role can be deduced from the mandatory role `@owner_att` since attributes and their owners must belong to identical schemas.

11.15 si_attribute

si_attribute		
$\xrightarrow{\text{isa}}$ attribute		
type	`char`	1–1
length	`integer`	1–1
decim	`integer`	0–1
stable	`integer`	0–1 (= `FALSE`)
recyclable	`integer`	0–1 (= `TRUE`)
@domain	0–1	`data_object`

Objects of type `si_attribute` represent simple attributes (*ie.* attributes whose the type is elementary — integer, string, character). The `type` field is the type of the attribute, this field is a character and special constants are available in `Voyager 2` to denote the different possible types:

`CHAR_ATT`: string with a constant length.

`VARCHAR_ATT`: string with a maximal length but variable.

`NUM_ATT`: numerical value with fixed integer and decimal parts.

`DATE_ATT`: date.

`BOOL_ATT`: boolean value.

`FLOAT_ATT`: real value with given precision.

`USER_ATT`: the domain of this attribute is predefined and it is found via the `domain` relationship.

`OBJECT_ATT`: the domain of this attribute is an entity-type[1] found it is found via the `domain` relationship.

`INDEX_ATT`: The value of the corresponding attribute is automatically computed by the DBMS in order that all the values are distinct (without gap).

`SEQ_ATT`: The value of the corresponding attribute is automatically computed by the DBMS in order that all the values are distinct.

[1] Although the `data_object` is the object type found on the other side of the `domain` relationship, only the entity-type instances are pertinent

When the user associates a *user domain* with a simple attribute, he attaches it a data object. This one is store in a special schema that you can retrieve with the following query:

```
GetFirst(SCHEMA[sch]{sch.name=SCHEMA_DOMAINS})
```

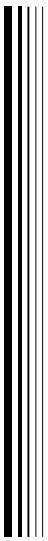
The user should use this schema to add new user defined domains. They will automatically appear in the ad-hoc dialog boxes of DB-MAIN.

The `length` field denotes the size of the value and the `decimal` field denotes the precision of the decimal type. The following table indicates when the `length` and `decimal` are required. Cells marked with a cross mean that a value is expected if the corresponding type is mentioned.

	length	decimal
CHAR_ATT	⋆	
VARCHAR_ATT	⋆	
NUM_ATT	⋆	⋆
DATE_ATT	10	
BOOL_ATT	1	
FLOAT_ATT	⋆	
USER_ATT		
OBJECT_ATT		

The `stable` field denotes an attribute that can not change once it received a value. The `recyclable` field denotes an attribute whose the value can be reused.

To create a new `si_attribute`:



```
what:=create(SI_ATTRIBUTE,
    name:string,
    min_rep:integer,
    max_rep:integer,
    type:char,
    length:integer,
    short_name:string,
    stable:integer,
    recyclable:integer,
    container:char,
    decim:integer,
    sem:string,
    tech:string,
    where:attribute,
    flag:integer,
    @OWNER_ATT:owner_of_att,
    @DOMAIN:data_object);
```

The `where` tells where the new attribute must be placed among all its new brothers. If this information is not supplied, the new attribute becomes the first attribute of the father (an entity-type, a rel-type or a compound attribute). Otherwise, the information must denote an attribute whose the father is identical to the one specified by the field found after `@OWNER_ATT`. So, to create a simple attribute which references an entity type (`ent`) we could write:

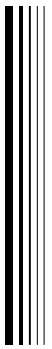
```
si_attr:=create(SI_ATTRIBUTE,type:OBJECT_ATT,min_rep:1,
max_rep:MAX_CON,container:BAG_ATT,@OWNER_ATT:the_owner,@DOMAIN:ent);
```

11.16 co_attribute

co_attribute
$\xrightarrow{\text{isa}}$ attribute
$\xrightarrow{\text{isa}}$ owner_of_att

Objects of type `co_attribute` represent compound attributes. The type of such attributes depends on the types of the components, therefore only the `min_rep` and `max_rep` fields must be mentioned in the environment as well as the name (of course) and the owner.

To create new composed attributes:



what:=create(`CO_ATTRIBUTE`,
`name`:*string*,
`short_name`:*string*,
`min_rep`:*integer*,
`max_rep`:*integer*,
`container`:*char*,
`sem`:*string*,
`tech`:*string*,
`where`:*attribute*,
`flag`:*integer*,
`@OWNER_ATT`:*owner_of_att*);

See 11.15 for more details about the `where` field.

11.17 owner_of_att

owner_of_att		
$\xrightarrow{\text{isa}}$ generic_object		
owner_att	0–∞	`attribute`

The type `owner_of_att` has no creator. Objects of this type are just a generalization of objects of `ent_rel_type` and `co_attribute` types. The semantics of such objects simply means that they can own attributes.

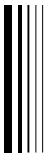
11.18 component

component		
$\xrightarrow{\text{isa}}$ generic_object		
@real_comp	1–1	`real_component`
@gr_comp	1–1	`group`

Objects of `component` type are just artifacts to represent many-to-many relationships. Therefore there are no other information in the environment than both objects playing a role in the many-to-many relationship. However, since objects order is of importance, a special field is used to denote the component

preceding the new one in the set of components owned by a group. If this information is not provided in the environment, then the new component becomes the first one of the group. This information is supplied after the constant `where`.

To create a new object of type `component`:



what:=create(COMPONENT,
@REAL_COMP:*real_component*,
@GR_COMP:*group*,
flag:*integer*,
where:*component*);

For instance, let us suppose that the attribute `at` must be inserted at the second place in the group `gr` then, if the first item in this group is linked to the group by the component `co`, the instruction will be:

```
new_one:=create(COMPONENT,@REAL_COMP:at,@GR_COMP:gr,WHERE:co);
```

11.19 group

6.0

group		
$\xrightarrow{\text{isa}}$ user_object $\xrightarrow{\text{isa}}$ owner_of_proc_unit		
name	string [L_NAME]	1–1
type	char	0–1 (= ASS_GROUP)
primary	integer	0–1 (= 0)
secondary	integer	0–1 (= 0)
coexistence	integer	0–1 (= 0)
exclusive	integer	0–1 (= 0)
atleastone	integer	0–1 (= 0)
key	integer	0–1 (= 0)
constraint	integer	0–1 (= 0)
funct	integer	0–1 (= 0)
min_rep	integer	0–1 (= 0)
max_rep	integer	0–1 (= 0)
sem	string	0–1 (= " ")
tech	string	0–1 (= " ")
@data_gr	1–1	data_object
gr_mem	0–∞	member_cst

6.0

A group in the DB-MAIN's repository is a set of properties like attributes, roles and groups themselves. Elements of groups have their own type: `real_component` that is the supertype of the previous types. A group can be attached to an entity-type, a rel-type or an attribute, therefore there is a link between `data_object` and `group`.

The fields of a group are described below:

name: The group name is mandatory but will often be a technical name. All the groups attached to the same data_object must have distinct names.

type: The type is either `ASS_GROUP` or `COMP_GROUP`.

primary/secondary: The primary and secondary fields denote respectively primary and secondary identifiers and must be used as boolean values.

coexistence: This field is used as a boolean value to indicate if all the items present in the group must be instantiated together or not.

exclusive: This field is used as a boolean value to indicate that at most one item in the group can be instantiated.

atleastone: This field is used as a boolean value to indicate that at least one item in the group must be instantiated.

key: The group is used as a key to access the object containing the group.

constraint: This field is used as a boolean value to indicate that the `meta_property` "Constaint" contains the user contraint type. ***6.0***

funct: This field is an array of bits that stores the different functions of the group (primary, secondary, coexistence, exclusive, atleastone, key, constraint). The corresponding constants are `ID_GR`, `SEC_GR`, `COEX_GR`, `EXCL_GR`, `AL1_GR`, `KEY_GR` and `CST_GR`. ***6.0***

To create a new group, use the instruction:

```
what:=create(GROUP,
    name:string,
    type:char,
    primary:integer,
    secondary:integer,
    coexistence:integer,
    exclusive:integer,
    atleastone:integer,
    key:integer,
    constraint:integer,
    funct:integer,
    sem:string,
    tech:string,
    flag:integer,
    @DATA_GR:data_object);
```

11.20 constraint

constraint		
$\xrightarrow{\text{isa}}$ generic_object		
type	`char`	1–1
const_mem	0–∞	`member_cst`

6.0

Let us note that only the `type` field is required in the environment of this object. Although its domain is a character, its possible values are restricted to five constants: `EQ_CONSTRAINT`, `INC_CONSTRAINT`, `INV_CONSTRAINT`, `GEN_CONSTRAINT` and `INCLUSION_CONSTRAINT`. ***6.0***

A constraint is a property attached to a tuple of groups $(g_1, g_2, \ldots, g_n)$. Each component of the tuple plays a special part for the constraint. The name of the role is specified in the object `member_cst`. The type of the constraint is stored in the `type` field of `constraint`. Only two types of constraints are defined so

far: INC_CONSTRAINT and EQ_CONSTRAINT. They are both binary and their roles have the same names: OR_MEM_CST (for *origin*) and TAR_MEM_CST (for *target*).

Let us suppose that the group g_1 (composed of the *name* and *first_name* attributes of an entity e_1) is a foreign key to a group g_2 (composed of the *N_F_names* attribute). Then g_1 will be the *origin* of one inclusion constraint and the *target* role is the group g_2. All the types of constraints respect the pattern explained here above.

INC_CONSTRAINT : Let g_1 and g_2 be two groups playing respectively the *origin* and *target* roles for the constraint. Then all the components of g_1 (attributes and/or roles) must take their values in the domain built on the values of all the components of g_2.

EQ_CONSTRAINT : Let g_1 and g_2 be two groups participating in an *equality* constraint, then the *inclusion* constraint holds for both (g_1, g_2) and (g_2, g_1). Therefore, this constraint should not be oriented: g_1 and g_2 play the same role. However, some physical models require that a group must be the origin of the other one[1].

⌜6.0⌝

INCLUSION_CONSTRAINT : the constraint is an inclusion if each instance of the first group must be an instance of the second group (the second group must not be an identifier, the inclusion constraint is the generalization of the referential constraint).

⌞6.0⌟

⌜6.0⌝

INV_CONSTRAINT : the constraint means that each object attribute is the inverse ot the other.

⌞6.0⌟

⌜6.0⌝

GEN_CONSTRAINT : the constraint is a generic constraint between any groups (without preconditions).

⌞6.0⌟

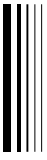
To create a new constraint:

what:=create(CONSTRAINT,
type:*char*,
flag:*integer*);

11.21 member_cst

member_cst		
$\xrightarrow{\text{isa}}$ generic_object		
mem_role	string [L_ROLE]	0–1 (= "")
@const_mem	1–1	constraint
@gr_mem	1–1	group

See section 11.20 for more details. To create a new member_cst object:



what:=create(MEMBER_CST,
mem_role:*string*,
flag:*integer*,
@CONST_MEM:*constraint*,
@GR_MEM:*constraint*);

[1] This constraint is usually implemented in SQL by a foreign key followed by a check. The orientation takes a sense here!

11.22 collection

collection		
$\xrightarrow{\text{isa}}$ user_object		
name	`string` [L_NAME]	1–1
short_name	`string` [L_SNAME]	0–1 (= " ")
sem	`string`	0–1 (= " ")
tech	`string`	0–1 (= " ")
@sch_coll	1–1	`schema`
coll_colet	0–∞	`coll_et`

Objects of type `collection` denote files, clusters, areas ... Each collection is identified by its name and a schema (`sch_coll` link).

To create a new object of type `collection`:

```
what:=create(COLLECTION,
    name:string,
    short_name:string,
    sem:string,
    tech:string,
    flag:integer,
    @SCH_COLL:schema);
```

11.23 coll_et

coll_et		
$\xrightarrow{\text{isa}}$ generic_object		
@coll_colet	1–1	`schema`
@data_colet	1–1	`data_object`

Objects of this type are technical instances to represent many-to-many relations between `collection` and `data_object` entities. Therefore the creation of these objects is derived from the attachment of an entity-type to a collection.

11.24 cluster

cluster		
$\xrightarrow{\text{isa}}$ user_object		
name	`string` [L_NAME]	0–1 (= " ")
total	`integer`	1–1
distinct	`integer`	1–1
criterion	`string` [L_CRITERION]	0–1 (= " ")
@entity_clu	1–1	`entity_type`
clu_sub	0–∞	`sub_type`

Objects of type `cluster` denote groups of distinct entity types being specialization of a super-type entity-type. The `total` and `disjoint` fields are boolean values characterizing the semantics of the group:

`total:` is `TRUE` if the set of all the instances of the super-type must be the same as the union of all the sets of instances of sub-types and `FALSE` otherwise.

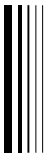
`disjoint:` is `TRUE` if to each instance of the super-type does correspond at most one instance in the range of its sub-types.

11.25 sub_type

sub_type		
$\xrightarrow{\text{isa}}$ generic_object		
value	`string` [L_VALUE]	0–1 (= " ")
@clu_sub	1–1	`cluster`
@entity_sub	1–1	`entity_type`

The object-type `sub_type` is due to the decomposition of one many-to-many relationship into two links (one-to-many). The `value` field is the criterion value upon which sub-types may be distinguished.

To create a new object of type `sub_type`



what:=create(`SUB_TYPE`,
`value`:*string*,
`flag`:*integer*,
`@CLU_SUB`:*cluster*,
`ENTITY_SUB`:*entity_type*);

11.26 role

role		
$\xrightarrow{\text{isa}}$ user_object		
name	`string` [L_NAME]	1–1(*cfr.* †)
min_con	`integer`	1–1
max_con	`integer`	1–1
sem	`string`	0–1 (= "")
tech	`string`	0–1 (= "")
@rt_ro	1–1	`rel_type`
ro_etr	0–∞	`et_role`

† Objects of type `role` denote roles of relationship types. Each role is identified by its name, its connectivities (min/max) and the relationship type it depends on. Although the name is mandatory, this field can be omitted if the following constraints are satisfied:

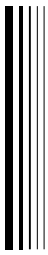
- the role is not a multi-entity role

$$\forall r \in \texttt{ROLE}[\ldots]\{\texttt{TRUE}\} : \texttt{Length}(\texttt{ET\_ROLE}[\ldots]\{@\texttt{RO\_ETR} : [r]\}) = 1$$

- the name of the entity type that should play the new role, must not be a name of another role of this relationship type.

If these constraints are satisfied, one considers that the name of the role is the name of the entity-type playing the role.

To create a new role:



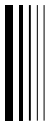
what:=create(ROLE,
name:*string*,
min_con:*integer*,
max_con:*integer*,
sem:*string*,
tech:*string*,
flag:*integer*,
@RT_RO:*rel_type*);

11.27 et_role

et_role		
$\xrightarrow{\text{isa}}$ generic_object		
@entity_etr	1–1	entity_type
@ro_etr	1–1	role

Each et_role object denotes a tuple (e, r) where e is an entity type and r is a role. Each tuple (e, r) means that the entity type e participates in the role r. If several entity types participate in a role r, we say that r is a *multi-ET*.

To create a new et_role:



what:=create(ET_ROLE,
flag:*integer*,
@ENTITY_ETR:*entity_type*,
@RO_ETR:*role*);

11.28 real_component

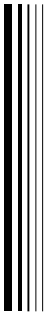
real_component		
$\xrightarrow{\text{isa}}$ generic_object		
real_comp	0–∞	component

The real_component type is just a supertype of the attribute, role and group types. Its only role in the repository is to denote a group item.

11.29 proc_unit

proc-unit		
$\xrightarrow{\text{isa}}$ data_object		
mode	char	0–1 (= unknown)
type	char	0–1 (= unknown)
@owner_pu	1–1	owner_of_proc_unit
pinvolves	0–∞	pinvolve
pbody	0–∞	pstatement
pinvokes	0–∞	pstatement
pfctcall	0–∞	pexpression
pdecl	0–∞	penvironment

This object represent a process (*in the large*). ⌜**6.0**⌝



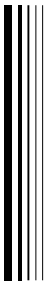
what:=create(PROC_UNIT,
`name`:*string*,
`short_name`:*string*,
`type`:*char*,
`sem`:*string*,
`tech`:*string*,
`flag`:*integer*,
`@SCH_DATA`:*schema*,
`@OWNER_PU`:*owner_of_proc_unit*
);

⌞**6.0**⌟

11.30 p_statement

p_statement		
$\xrightarrow{\text{isa}}$ generic_object		
type	`char`	1–1
description	`string`	0–1 (= `" "`)
@p_body	0–1	`proc_unit`
@p_invokes	0–1	`proc_unit`
p_made_of	0–∞	`proc_unit`
p_part_of	0–∞	`proc_unit`

This object represents a statement. At least one of the both roles `@p_body` and `@p_invokes` are mandatory. The `where` attribute indicates the left brother of the new object in the relation that comes afterwards.



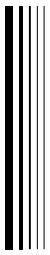
what:=create(P_STATEMENT,
`type`:*char*,
`description`:*string*,
`flag`:*integer*,
`where`:*p_statement*,
`@P_BODY`:*proc_unit*,
`where`:*p_statement*,
`@P_INVOKES`:*proc_unit*
);

11.31 p_component

p_component		
$\xrightarrow{\text{isa}}$ generic_object		
mode	`char`	0–1 (= 0)
type	`char`	0–1 (= 0)
@p_made_of	1–1	`p_statement`
@p_part_of	1–1	`p_statement`

This object denotes the decomposition of a statement into its sub-components. The type and mode attributes denote the kind of decomposition. The `where`

attribute indicates the left brother of the new object in the relation that comes afterwards.

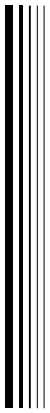


what:=create(P_COMPONENT,
`type`:*char*,
`mode`:*char*,
`flag`:*integer*,
`where`:*p_component*,
`@P_MADE_OF`:*p_statement*,
`where`:*p_component*,
`@P_PART_OF`:*p_statement*

11.32 p_expression

p_expression		
$\xrightarrow{\text{isa}}$ generic_object		
operator	`char`	1–1
constant	`string`	0–1 (= `""`)
description	`string`	0–1 (= `""`)
@p_sub_expression_of	0–1	`p_expression`
p_sub_expression_of	0–∞	`p_expression`
@p_parameter	0–1	`p_statement`
@p_fct_call	0–1	`p_proc_unit`
@p_act_arg	0–1	`generic_object`

This object denotes an expression. At least one of the four possible roles must be defined in the creator and the `where` argument that precedes a role indicates the place of the new object amongst its "brothers" inside the relation.

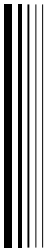


what:=create(P_EXPRESSION,
`operator`:*char*,
`constant`:*string*,
`description`:*string*,
`flag`:*integer*,
`where`:*p_expression*,
`@P_SUB_EXPRESSION`:*p_expression*,
`where`:*p_expression*,
`@P_ACT_ARG`:*generic_object*,
`where`:*p_expression*,
`@P_PARAMETER`:*p_statement*,
`where`:*p_expression*,
`P_FCT_CALL`:*proc_unit*,

11.33 p_environment

p_environment		
$\xrightarrow{\text{isa}}$ generic_object		
type	`char`	1–1
mode	`char`	0–1 (=0)
@p_decl	1–1	`proc_unit`
@p_go_env	1–1	`generic_object`

The `where` argument defines the place of the newly created object in the one-to-many relation that comes just afterwards. If this argument is not present, the new object is inserted at the first place in the relation.

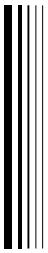


what:=create(P_ENVIRONEMNT,
`type`:*char*,
`mode`:*char*,
`flag`:*integer*,
`where`:*p_environment*,
`@P_DECL`:*proc_unit*,
`where`:*p_environment*,
`@P_GO_ENV`:*generic_object*

11.34 p_involve

p_involve		
$\xrightarrow{\text{isa}}$ generic_object		
type	`char`	0–1 (= 0)
mode	`char`	0–1 (= 0)
@p_involved_in	1–1	`p_actor`
@p_involved	1–1	`proc_unit`

The `where` argument defines the place of the newly created object in the one-to-many relation that comes just afterwards. If this argument is not present, the new object is inserted at the first place in the relation.

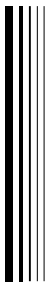


what:=create(P_INVOLVE,
`type`:*char*,
`mode`:*char*,
`flag`:*integer*,
`where`:*p_involve*,
`@P_INVOLVED_IN`:*p_actor*,
`where`:*p_involve*,
`@P_INVOLVES`:*proc_unit*

11.35 p_function

p_function		
$\xrightarrow{\text{isa}}$ generic_object		
type	`char`	0–1 (= 0)
mode	`char`	0–1 (= 0)
description	`string`	0–1 (= "")
@p_has_a	1–1	`p_actor`
@p_attributed_to	1–1	`p_actor`

The `where` argument defines the place of the newly created object in the one-to-many relation that comes just afterwards. If this argument is not present, the new object is inserted at the first place in the relation.



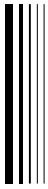
what:=create(P_FUNCTION,
name:*string*,
type:*char*,
mode:*char*,
flag:*integer*,
where:*p_function*,
@P_HAS_A:*p_actor*,
where:*p_function*,
@P_ATTRIBUTED_TO:*p_actor*

11.36 p_actor

p_actor		
$\xrightarrow{\text{isa}}$ generic_object		
name	string	1–1
sem	char	0–1 (= "")
tech	char	0–1 (= "")
p_involved_in	0–∞	p_involve
p_has_a	0–∞	p_function
p_attributed_to	0–∞	p_function

The where argument denotes the system the new object will belong to. So far, there is a unique system. The programmer can retrieve it with this expression:

```
GetFirst(SYSTEM[sys]TRUE)
```



what:=create(P_INVOLVE,
name:*string*,
sem:*string*,
tech:*string*,
flag:*integer*,
where:*system*

11.37 owner_of_proc_unit

owner_of_proc_unit		
owner_pu	0–∞	proc_unit

11.38 meta_object

meta_object		
$\xrightarrow{\text{isa}}$ generic_object		
name	string [L_NAME]	1–1
type	integer	1–1
sem	string	0–1 (= "")
@sys_mo	1–1	system
mo_mp	0–∞	meta_property

To each instance of `meta_object` corresponds exactly one object type of the repository. The aim of each meta-object –ie: instance– is to describe the corresponding object type. So far, this object type should be *read-only*. In a near future, creation of new instances of this object type will have as consequence to extend the repository with new object types. You could add new object types like *agent*, *module*, ...

The repository part made of the `meta_object` and the `meta_property` (*cfr.* 11.39) object types is called "*the meta-definition*". This meta-definition will possibly evolve in the future as explained early.

Only the objects that are not a super-type of another object type and that does not belong to the meta-definition are described by the meta-definition.

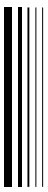
A description of the fields follows:

`name:` It is the object type's name ("entity_type", "system", "si_attribute", ...).

`sem:` It is the semantics description.

`type:` It is an integer constant identifying the type of the object type. These constants are `ENTITY_TYPE`, `GROUP`, ... (*cfr.* 2.4).

Each time one loads/creates one project, these instances are created by `DB-MAIN`. So you should not create such objects yourself.



what:=create(META_OBJECT,
`name`:*string*,
`type`:*integer*,
`sem`:*string*,
`flag`:*integer*,
`@SYS_MO`:*system*);

11.39 meta_property

meta_property		
$\xrightarrow{\text{isa}}$ generic_object		
name	`string` [L_NAME]	1–1
type	`integer`	1–1
updatable	`integer`	0–1 (= `TRUE`)
predefined	`integer`	0–1 (= `FALSE`)
multi	`integer`	0–1 (= `FALSE`)
hidden	`integer`	0–1 (= `FALSE`)
sem	`string`	0–1 (= `""`)
@mo_mp	1–1	`meta_object`

If an object type is described by one instance of `meta_object`, it is possible to dynamically add new fields to it (*cfr.* 15.2 for more details). These fields are named *dynamic properties*. Each dynamic property is described by one instance of the `meta_property` object type.

A description of the fields follows:

`name:` The name of the property.

`type`: The type of the property. Constants explained in 11.15 are used to denote the type (`string`: `VARCHAR_ATT`, `char`: `CHAR_ATT`, `integer`: `NUM_ATT`, `float`[1]: `FLOAT_ATT`, `boolean`: `BOOL_ATT`).

`updatable`: This field is an integer value used as a boolean value. If the value is equivalent to the constant `TRUE`, then the meta-property may be updated in the `DB-MAIN` environment. Otherwise, the meta-property can only be read/consulted and can not be edited/modified. However, `Voyager 2` programs can always edit this field whatever it is *updatable* or not.

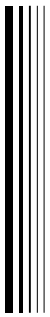
`multi`: This field is an integer value used as a boolean value. If this field has the value true, the meta-property may be multivalued. The value of such a meta-property is then a list of values whose the type is the one indicated by the field `type` described above.

`predefined`: This field is an integer value used as a boolean value. If this field has the value true, then the possible values of such a meta-property *should* occur in a predefined list of values. For instance, a meta-property called `sex` could have as predefined value `female` or `male`. The predefined values must be compliant with the type of the meta-property and are stored in the semantics description of the meta-property as a *textual property*. The user is obliged to choose one of these values and has no other choice. This restriction only holds in the `DB-MAIN` tool but is not true for a `Voyager 2` program. There is no automatic validation in `Voyager 2`. This responsability is let to the programmer.

`hidden`: This field is an integer value used as a boolean type. When it is worth `TRUE`, the meta property is predefined and should be used very carefully.

`sem`: The semantics description of the dynamic property.

Each meta-property is attached to the meta-object it describes by the link: `MO_MP`.



what:=create(META_PROPERTY,
`name`:*string*,
`type`:*integer*,
`sem`:*string*,
`updatable`:*integer*,
`predefined`:*integer*,
`multi`:*integer*,
`hidden`:*integer*,
`flag`:*integer*,
`@MO_MP`:*meta_object*);

The next sample shows how creating meta-properties.

```
meta_property:  mp;
meta_object:  mo;
entity_type:  ent;

begin
    // add the meta-property 'local' to each collection
```

[1]This type is not yet supported by `Voyager 2`.

```
    // the meta-property is predefined and the allowed values are:
    //   Bruxelles, Paris, Madrid, London
    mo:=GetFirst(META_OBJECT[mo]{mo.type=COLLECTION});
    mp:=create(META_PROPERTY,name:"local",type:VARCHAR_ATT,
          predefined:1,@MO_MP:mo);
    mp.sem:=GetProperty(mp.sem,
          "Bruxelles\nParis\nMadrid\nLondon");
    // add the meta-property 'owners' to each entity-type
    // the meta-property is multivalued
    mo:=GetFirst(META_OBJECT[mo]{mo.type=ENTITY_TYPE});
    mp:=create(META_PROPERTY,name:"local",type:VARCHAR_ATT,
          multi:1,@MO_MP:mo);
    ...
    // let 'ent' be an entity-type
    // add 'tintin' to the list of owners.
    ent."owners":=ent."owners"++["tintin"];
    print(ent."owners");
end
```

11.40 user_viewable

user_viewable		
object_view	0–∞	user_view

11.41 user_view

user_view		
$\xrightarrow{\text{isa}}$ generic_object		
name	string	1–1
type	char	1–1
font_name	string	1–1
font_size	integer	1–1
mark_plan	integer	1–1
reduce	integer	1–1
text_font_name	string	1–1
text_font_size	integer	1–1
xgrid	integer	1–1
ygrid	integer	1–1
zoom	integer	1–1
@object_view	1–1	user_viewable
uv_uo	0–∞	user_object

This object denotes the graphical representation of a product (schema, document, or a system). The programmer is not allowed to create new user views.

The font_name and font_size attributes denotes the font used in graphical views. The text_font_name and text_font_size attributes denotes the font used in textual views. The mark_plan attribute is the current mark plan used in a schema (see 79 for more details). The reduce attribute is the current reduce factor (per cent) in the graphical views. The zoom attribute is the current zoom factor (per cent) in the graphical views. The xgrid and ygrid attributes denote the size of the page drawn in the graphical views.

11.42 product_type

product_type		
$\xrightarrow{\text{isa}}$ generic_object		
title	`string`	1–1
min_mul	`integer`	1–1
max_mul	`integer`	1–1
description	`string`	0–1 (= "")

This object is used to give a type to products created by `Voyager 2` procedures to be used with an MDL method [7]. `product_type` are read-only objects: their fields an be read but their values cannot be changed and new `product_type` cannot be created.

A description of the fields follows:

`title`: the name of the product type.

`min_mul`: the minimum number of products that should be created with this type.

`max_mul`: the maximum number of products that should be created with this type.

`description`: a small text describing the semantics of the product type.

Note that `min_mul` and `max_mul` are guidelines which are not enforced.

11.43 schema_type

schema_type
$\xrightarrow{\text{isa}}$ generic_object

A type for schema created within an MDL method [7].

11.44 document_type

document_type
$\xrightarrow{\text{isa}}$ generic_object

A type for document created within an MDL method [7]. ⌞***6.0***⌟

Chapter 12

Predicative Queries

12.1 Introduction

Although navigational queries are provided in `Voyager 2` (see page 111), predicative queries are possible and welcome in `Voyager 2` since this kind of query is the most used. The aim of this query is to hide and to factorize boring and technical details when querying the repository. With predicative queries, the programmer has not to tell how to obtain a result but simply what he wants. Another characteristic is the guaranty to keep the same performance than a *hand-written* algorithm.

For instance: to get all the optional attributes from a project, the following query will return the expected result in a list:
Example:

```
ATTRIBUTE[att]{att.min_rep=0}
```

□
This query is of course an expression and it can be used everywhere a list-expression is expected. Another form of query is illustrated here:
Example:

```
for dat in DATA_OBJECT[dat]{@SCH_DATA:[GetCurrentSchema()]}

do {

  ⋮

};
```

□
this example shows how to iterate through all the data-objects from the current schema. And finally a more complex query:
Example:

```
list_result:=DATA_OBJECT[dat]{@SCH_DATA:[GetCurrentSchema()]
with soundex(dat.name,"bank")};
```

□
where the result is all the data-objects from the current schema having a name similar to "bank"[1].

Predicative queries have two forms: *global scope* and *restricted scope* queries. A formal specification is given in the following sections.

12.2 Specifications

Queries always respect the following syntax:

⟨*query*⟩ ← ⟨*ent-expr*⟩ "[" ⟨*variable*⟩ "]" "{" ⟨*constraint*⟩ "}"

where ⟨*ent-expr*⟩ is an integer expression denoting an object of the repository. Although any integer expression is valid, programmers will usually use constants from the table 2.4. The meaning of each constant has been explained in the chapter 11. ⟨*variable*⟩ denotes a variable whose the type must be exactly the same as the object type represented by the ⟨*ent-expr*⟩ expression[2]. ⟨*constraint*⟩ is used to sort out the pertinent objects from the others.

Once the query has been evaluated, the value of the ⟨*variable*⟩ is undefined. During the evaluation, the program must not modify its content. The scope of the ⟨*variable*⟩ may be just a portion of the ⟨*constraint*⟩. This last characteristics will be explained here below. The ⟨*variable*⟩ is called the *iterator* for convenience.

12.2.1 Global Scope Queries

Global-scope queries look the whole repository for objects satisfying the constraint. The constraint may be any integer expression but is used as a boolean value. The whole constraint is in the scope of the iterator. Thus the syntax of the constraint is:

⟨*constraint*⟩ ← ⟨*boolean-constraint*⟩
⟨*boolean-constraint*⟩ ← ⟨*integer-expression*⟩

For instance:
Example:

```
ENTITY_TYPE[e]{TRUE}
```

□
will look for all the entity-types of the project. All these entity-types are then stored in a list. The following example shows how to use the variable specified in the query to express more accurate constraints:
Example:

[1] `soundex` is not a primitive of `Voyager 2`.

[2] Although entity type may be an expression and thus may vary at the execution time, the variable must be typed at the compilation time and can thus not vary during the execution. This characteristics may seem strange. Why is the compiler not able to deduce the type of the object from the variable? First, variables names are not always significant and therefore queries would be difficult to understand. And finally, this feature could be used in a more advanced version of `Voyager 2`.

```
ATTRIBUTE[a]{a.min_rep=1 and a.max_rep=1}
```

□

The query looks for all the single-valued mandatory attributes.

Objects having one or more sub-types can not be used in *global scope* queries! The query will return an empty list for such cases! This restriction is present since the 1.0 release but this constraint was omitted in the previous manuals.

12.2.2 Restricted Scope Queries

In restricted-scope queries, constraints have two components: the *link-constraint* and the *boolean-constraint*. Although the first part is mandatory, the second one is not. The syntax of this constraint is:

⟨*constraint*⟩ ← ⟨*link-constraint*⟩ ["with" ⟨*boolean-constraint*⟩]
⟨*link-constraint*⟩ ← ⟨*link-expr*⟩ : ⟨*list-fathers*⟩
⟨*boolean-constraint*⟩ ← ⟨*integer-expression*⟩
⟨*link-expr*⟩ ← ["@"] ⟨*integer-expression*⟩
⟨*list-fathers*⟩ ← any expression of type `list`

The ⟨*link-expr*⟩ is an integer expression denoting a link between two objects. Usually, programmers will use constants rather than complex expressions due to reasons explained in the previous section. The table 2.5 contains all the constants that the programmer could ever need. Each constant is explained in chapter 11. Because links are oriented, it suffices to reverse the sign of a ⟨*link-expr*⟩ to reverse the corresponding link. In queries, a special token "@" is equivalent to the unary operator: "−" and programmers are encouraged to use it in order to make the query more readable. Therefore, if L is a constant denoting a link, then we have: $L \equiv @@L$. This property will be used below.

The ⟨*ent-expr*⟩ expression denotes an object type (see above). This object type **must be exactly** the same as the one playing the role $@L$ if L is the value of "⟨*link-expr*⟩".

The ⟨*list-fathers*⟩ expression denotes any expression for which the evaluation will return a list. Let us note L, the value returned by the evaluation of "⟨*link-expr*⟩". Then each item of the list l must be compatible with the object type playing the role $@L$. **The ⟨*list-fathers*⟩ is not in the scope of the iterator.**
Example:

> If L is worth `@PFROM`, then the object type playing the role `@@PFROM` (≡`PFROM`) is `product`. All the compatible object types with `product` are: `document`, `schema` and `generic_object`[1].

□

[1] Let us note that the last object type is slightly different from the previous ones since all the generic objects are not necessarily one product although each document/schema is a product. Thus if your list contains generic objects, you must have the insurance that they are all compatible with the `product` object type. Otherwise, it is a bug!

The ⟨*boolean-constraint*⟩ must be preceded by the `with` keyword. This part is optional in the constraint. This expression is in the scope of the iterator and can thus use it.

To make easier our examples, we will use a small "academic" schema definition. Let us suppose that A, A_0, A_1, B, B_0 and B_1 are new object types. We note a, a_0, a_1, b, b_0 and b_1 their respective fields. The following relations hold: A_1 `isa` A, A `isa` A_0, B_1 `isa` B and B `isa` B_0. There is a link noted L between A and B: $A \xrightarrow{L} B$. The schema is depicted in figure 12.1.

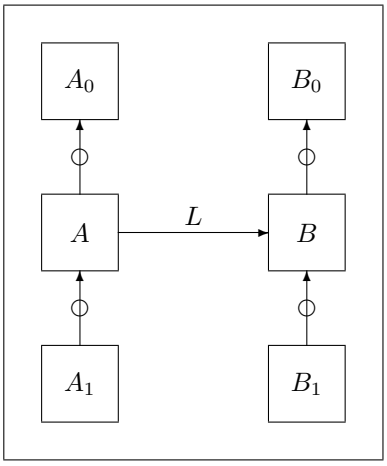


Figure 12.1: Acadamic Schema.

We give here some examples as exercices. The solutions are given just after. Are these requests correct?

Examples:

1) $A[\alpha]\{L : [\beta_1, \beta_2]\}$
where $A : \alpha;\ B : \beta_1,\ \beta_2;$

2) $A_0[\alpha_0]\{L : [\beta_1, \beta_2]\}$
where $A_0 : \alpha_0;\ B : \beta_1,\ \beta_2;$

3) $A[\alpha]\{L : [\beta_0, \beta_1]\}$
where $A : \alpha;\ B_0 : \beta_0;\ B_1 : \beta_1;$

4) $B : [\beta]\{@L : A[\alpha]\{\alpha.a$`>=`$3\}$ `with` $\beta.b$`<=`5 `and` $\beta.b$`>`$0\}$
where $A : \alpha;\ B : \beta$

5) $A[\alpha]\{L : B[\beta]\{@L : A[\alpha_1]\{\alpha_1.a = \beta.b\}$ `with` $\beta.b = 3\}\}$
where $A : \alpha, \alpha_1;\ B : \beta$

6) $A[\alpha]\{L : B[\beta]\{@L : A[\alpha]\{\alpha.a = 6\}$ `with` $\beta.b = 3\}\}$
where $A : \alpha;\ B : \beta$

7) $A[\alpha]\{L : B[\beta]\{@L : A[\alpha_1]\{\alpha_1.a = 6\}$ `with` $\beta.b = 3\}\}$
where $A : \alpha, \alpha_1;\ B : \beta$

8) $A[\alpha]\{L : B[\beta]\{\beta.b = 1\} + + B[\beta]\{@L : A[\alpha_1]\{\alpha_1.a = 2\}\}\}$
where $A : \alpha, \alpha_1;\ B : \beta$

Solutions

1) YES: without any comments!

2) NO: L is a role played by the A object type. You can not use A_0 in place of A in this query.

3) YES: The list of fathers $[\beta_0, \beta_1]$ may be composed of any value for which the type is compatible with the object type playing the role $@L$.

4) YES: without any comments!

5) NO: The constraint $\alpha_1.a = \beta.b$ is not in the scope of the β iterator. The value of β is undefined here.

6) NO: The same variable α is used two times as iterator in the query.

7) YES: without any comments!

8) YES: The query asks for all the object from A that play the role L for one element of the list $B[\beta]\{\beta.b = 1\} + +B[\beta]\{@L : A[\alpha_1]\{\alpha_1.a = 2\}\}$. This list is a new expression that is independent of the query. This expression is composed of two operands: two queries. Each query is computed, the result is a list. The both lists are appended and used as a "list of fathers" in the original query. Note: this query is not very efficient. How could you rewrite it more efficiently?

□

Chapter 13

Iterative Queries

Although predicative queries are very useful, they are sometimes not sufficient. For this reason, `Voyager 2` offers basic primitives to access to the content of the repository: `_GetFirst()`, `_GetNext()`, `TheFirst` and `TheNext`. Formal specifications follow.

function generic_object: *o* `TheFirst` (integer: *t*)

Precondition. *t* must be an integer expression. The evaluation of *t* must return a value amongst the constants of table 2.4 and must denote an object type.

Postcondition. *o* is the first object found within the object type denoted by *t*. If the object type *t* has no instances, then *o* is void.

function generic_object: *o* `TheNext` (integer: *t*, any: *p*)

Precondition. *t* must be an integer expression. The evaluation of *t* must return a value amongst the constants of table 2.4 and must denote an object type. *p* must be a reference to one object of the object type denoted by *t*. *p* must be different of void.

Postcondition. *o* is the object that follows *p* within the instances of the object type denoted by *t*. If *p* is the last object, then *o* is void.
on error: The behaviour is uncertain.

function any: *s* `_GetFirst` (integer: *l*, any: *f*)

Precondition. the value of *l* must denote a link (see table 2.5). *f* must be different of void and its type must be compatibe with the object type that plays the role *l*.

Postcondition. If $[s_1,\ldots,s_n]$ are all the sons of the link *l* when *f* plays the role *l*, then *o* is the first element of this list. If the list is empty, then *s* is void. The type of *o* is the type of the object type that plays the role @*l*.
on error: The behaviour is uncertain.

function any: b _GetNext (integer: l, any: f, any: s)

Precondition. the value of l must denote a link (see table 2.5). f must be different of void and its type must be compatibe with the object type that plays the role l. s must be one of the sons of the link l when f plays the role l and the type of s must be compatibe with the object type that plays the role $@l$.

Postcondition. Let $[s_1, \ldots, s_i, s_{i+1}, \ldots, s_n]$ be the list of all the sons of the link l when f plays the role l. If the precondition is verified, then $\exists i : \ s_i = s$. If $i < n$ then $o = s_{i+1}$, and if $i = n$ then $o =$void.
on error: The behaviour is uncertain.

One recommends to avoid the use of these functions as much as possible since predicative queries have the same performances and are less error prone.
Example:

This example shows how to use the _GetNext function:

```
entity_type:  ent;
attribute:  att;
begin
     ent:= one entity type expression;
     for att in ATTRIBUTE[att]{@OWNER_ATT:[ent]}
     do {
        print(att.name);
        if IsNoVoid(_GetNext(OWNER_ATT,ent,att)) then {
          print(',');
        };
     };
end
```

□

Chapter 14

Object Removal

The removal of one object from the database is done by a call to the `remove` procedure. This procedure accepts one argument: the object to remove. Although this procedure is very simple for the user, this task is very complex since it destroys a lot of useless objects that would no more satisfy the integrity constraints after the removal of the main object (the argument).

procedure `remove` (object: *o*)

Precondition. *o* may be any object of the database. This value must not be `void`.

Postcondition. The object is removed from the database. All the objects that depends on it are also removed. For instance, the removal of one entity-type will also remove all its attributes, its groups, its roles, ...

Example:

```
remove(GetFirst(ENTITY_TYPE[ent]{name="CLIENT"}));
```

□

Chapter 15

Properties

The repository of DB-MAIN is staticaly defined with C++ classes. For this reason, we could say that the repository definition can not be modified in a dynamic way. However, the definition let some doors opened for such extensions. At present at least two ways exist to extend the repository: **Textual Properties** and **Dynamic Properties**.

The two kinds of properties have the same aim: "*adding new fields to one object-type in the repository*". Although the aim is identical, technics to implement them are completely different, and we will enforce to explain in the next parts the difference between each one.

15.1 Textual Properties

Textual properties are new properties attached to one object in the repository. This information is stored in either the semantic description field or in the technical description field of the object. These properties are not completely supported in Voyager 2, but two functions exist to help the programmer to manage them: GetProperty and SetProperty. Each function is fully described here after. Let us begin by giving an example of a *property*. Let us suppose that *o* is one object (ex: one entity-type variable). Now, the technical description of this object is :

> "This object denotes a car bought by a firm.↩
> Each car is pink.↩
> eof"

If you wish to add a new property to some entity-types like: the average of instances in the database[1], you could represent this information in the technical description:

> "This object denotes a car bought by a firm.↩
> Each car is pink.↩
> #average=26↩
> eof"

[1] If your database is relational, then you will use "tuple" or "line" in place of instance.

So that you can retrieve easily this information by a lexical analysis of the text.

Conventions are defined in DB-MAIN to represent this kind of information inside texts. The definition is:

> A *textual properties* has two informations: the *field* and the *value*. The representation of this information can occur anywhere in a text with the following restrictions: the field is the list of characters found between the '#' character and the first occurrence of the '=' character. All the characters are significative and the interpretation is **case sensitive**. The '#' character must be preceded by a carriage-return or must be the first character of the text. The value associated to the field is the list of characters found just after the '=' character until the first '#' preceded by a carriage return or – otherwise– until the end of the text. In the first case, the sequence '↩#' does not belong to the value. If several properties exist in the text with the same field, then just the first occurrence is taken into account.

The two functions defined here after help the programmer to manage the textual properties stored in the semantic and technical description of any object (and in any text in general).

function string: *value* `GetProperty` (string: *s*, string: *field*)

Precondition. ∅

Postcondition. If the field *field* is found in the text *s*, then the associated value is returned to the user. Otherwise, the constant `PROP_NOT_FOUND` is returned. This message is distinct of any possible value! The text *s* is let unmodified. If the text is corrupted, then the message `PROP_CORRUPTED` is returned.

function string: *r* `SetProperty` (string: *s*, string: *field*, string: *value*)

Precondition. ∅

Postcondition. This function returns the string *s* where the *value* associated with the field *field* has been replaced by the text *value*. If the field were not present, the information is added at the end of the text.

Example:

```
schema:  sch;
integer:  i;
string:  s;
begin
   sch:=GetCurrentSchema();
   s:=GetProperty(sch.sem,"color");                    (1)
   if s=PROP_NOT_FOUND then { i:=0; }
   else { i:=StrStoi(s); };
   sch.sem=SetProperty(sch.sem,"color",StrItos(-i));
end                                                    (2)
```

At the point **(1)**, the description is:

"This schema will be printed on our printer with the color:↩
#color=4↩
#end↩
But if the color is negative, this color is used for the↩
background ! ↩
eof"

and after the execution of the program the semantic description of the schema has been updated with the inverse of the color field:

"This schema will be printed on our printer with the color:↩
#color=-4↩
#end↩
But if the color is negative, this color is used for the↩
background ! ↩
eof"

Let us note that the special line "#end" is used to mark the end of the property. This line could have been replaced by any property like this:

"#font=Arial↩
#end"

□

Last but not least, this last function retrieve all the properties from a string with their associated values and put them in a list that is returned to the user.

function list: l `GetAllProperties` (string: s)

Precondition. $\emptyset$

Postcondition. l is a list of pairs $[p_1, v_1, p_2, v_2, \ldots, p_n, v_n]$ where v_i is the value of a property called p_i. The p_i's are all the properties present in the text s.

15.2 Dynamic Properties

15.2.1 Introduction

Although *textual properties* already allow to extend the repository in a quite easy way. There exists another mechanism known as *dynamic properties*. The repository is partially[1] described in a subpart of the repository itself called "*meta-repository*". This subpart contains so far two entity types: `meta_object` and `meta_property`. To each entity type[2] of the repository corresponds one instance of the `meta_object` entity-type. In an analogous way, each attribute/property of one entity type is described by one instance of the `meta_property` entity type. The reader may observe a correspondence between this explanation and the behaviour of the relational systems (tables and system tables).

[1] In a near future, relationships and inheritance relationships will also be described in this meta subpart.

[2] This entity-type must not be a super-type.

This "meta"-description is very interesting since it permits to dynamically extend the repository. It suffices to add one new instance in the `meta_property` entity type and to link this one to one instance of `meta_object` in order to add one new property to the entity type described by the meta-object. One this mofication is done, the new property is available both in the CASE tool and in `Voyager 2`.

15.2.2 Explanation

One will use an example to illustrate the use of the dynamic properties. Let us suppose that in your firm, ER-schemas are defined by several people at the same time. To manage this complexity, you wish to add to each entity type and to each relationship of your schema a new field indicating who has added this object in the schema.

One way to proceed is in adding *textual properties*. You may add also *dynamic properties*. We will explained this last one.

First, we need to add one new property to the `entity_type` object. This property will represent a man/woman ("Albert", "Bill", "Jessica", ...) and thus its type will be `string`. The normal way would be to proceed in using the CASE tool from the menu: `File`→`Meta...`→`Properties`. The complicated way would be in using `Voyager 2`. And we will choose this last one since our purpose is to explain the dynamic properties in `Voyager 2`!

Let us begin by creating one new instance in `meta_property`:

```
meta_object:  mo;
meta_property:  mp;
begin
    mo := GetFirst(META_OBJECT[mo]{mo.type=ENTITY_TYPE});
    mp:=create(META_PROPERTY,name:"owner",type:VARCHAR_ATT,@MO_MP:mo);
...
end
```

You have right now added a new property named "owner" to the `entity_type` object. Each instance of this object type has thus a new field initialized with an empty string. A similar work should be done for the `rel_type` object type.

Let us suppose now that the entity type "CLIENT" was created by Mr Sherlock Holmes. You may use the new field to store the information.

```
entity_type:  ent;
...
    ent:=GetFirst(ENTITY_TYPE[ent]{ent.name="CLIENT"});
    ent."owner":="Sherlock Holmes";
    ...
    print([ent.name," has been created by ",ent."Owner",'\n']);
end
```

Let us remark that you can reference this field like any other field with two exceptions:

1. the name of the field is between double quotes. It is a string.
2. the machanism is not case sensitive.

The sentence found after the "." may be any expression. If the evaluation of the expression returns an integer, then the field is static, otherwise the value should be a string, and the field is *dynamic*. When the field is static — name, short_name, sem, tech, min_rep, ... —, the name of the field is predefined in `Voyager 2` as an integer constant.

Dynamic properties are much more efficient than textual properties and they can be edited directly in the CASE tool.

Part III

Modular Programming

Chapter 16

Library and process

One of the main innovations of the version 3.0 over the previous versions is the possibility to use `Voyager 2` programs as libraries or as process. This chapter explains how to use these new characteristics.

These new possibilities are due to a new and completely rewritten abstract machine. In the former versions, there were only one possible abstract machine running at once.

In the new architecture[1], several programs/abstract machines can occur at the same time, and one program can call another program or just one function from another program. This chapter explains how using these new capabilities.

16.1 The New Architecture

The abstract machine is composed of two memory blocks. The first one lodges the program code (ie. the `.oxo` program), the second one is the memory used during the execution of the program. They are respectively called the "image" and the "stack". In former version, there were only one image and one stack at once as already explained. The new architecture allows several images and stacks at once. Let us use the program "`foo.oxo`" as example. Once it is loaded, this program can be stored in one or more images (as many times it was loaded). Once a program is loaded, this program can be executed. This entails the creation of one stack to give a working space to the new process. But from the same image, another process can be runned that entails the creation of another stack, that is distinct of the first one. Thus, one have one image and two stacks. But the architecture allows to load another program, and hence, a new process, ie. another stack. The situation is depicted in figure 16.1.

The image stores all the "instructions" to be executed to run a program, and therefore each function/procedure has its representation inside the image. One could execute either the whole program (ie. the body) or simply one function/procedure. Let us remember that as long as a stack is preserved, the "memory" of the process is preserved. So, if one executes one process, this one

[1]The `call_V2` instructions that existed in some "draft" versions of `DB-MAIN` is now simulated. This instruction replaced the calling program with the new program, this is no more true. This instruction is now deprecated and should be avoided. It will be removed in the next version.

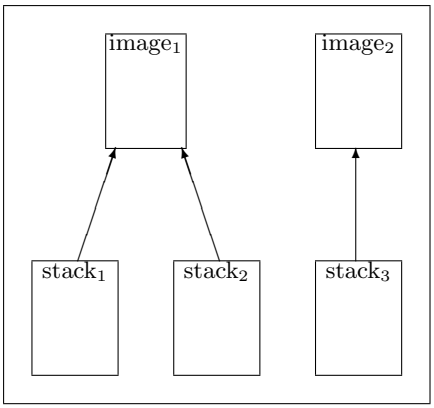


Figure 16.1: This schema depicts the memory state with two programs and three running process.

can let the global variables in some state (S) at the end of its execution, and retrieve the same state at the next execution.

We have now sufficient information to enter in more details in the mechanism. Two new types have been added to the language. The first one is `program`. A value of type `program` denotes a process, and one will see that once a value of this type is correctly initialized, one can start an execution of the associated program. The second type is `lambda`[1]. This one denotes an entry in the image of a program corresponding to a procedure or a function. Such a value, once initialized, can be used to start the execution of the associated function/procedure in using the stack of a process.

16.2 `Voyager 2` Process

The first step in the use of a process is the declaration of one variable of type `program`.

```
program:  p;
```

Once this variable is declared, you can initialize this variable with the `use` instruction.

```
p := use("c:\\ foo.oxo");
```

The `use` instruction has only one argument (one string) that denotes the program to be loaded into a newly created image. Once the `p` variable is initialized, you can call this program in such a way:

```
p!(a1,...,an);
```

[1]The "*lambda*" term derives from an area of mathematical logic called the "*lambda calculus*" in which many of the theorical foundations of functional languages are based [4], [2] . `lambda` expressions in `Voyager 2` have very few common characteristics with this concept but the author had not enough imagination to invent a new term.

The ! character is a suffix unary operator. The a_1, ..., a_n are values passed as arguments to the process. A new characteristic of `Voyager 2` is that a program can now return a value, and its type does not have to be specified. Hence, a program can either be executed as a procedure (an instruction) or a function (an expression) depending on the importance of the returned value in the calling program[1]. So, another call to the program `foo` could have been:

```
v := p!(a1,...,an);
```

We mentioned in the beginning of the chapter that functions/procedures could be called separately. The first step to call such a procedure is to get an "handle" of the function from the program/process. This is done with the binary operator `::`. The left operand denotes the process, and the right operand is a string that denotes the name of the function/procedure. The result of this operator is a value of type `lambda`. This result should obviously be stored in a variable to be used later. This can be done as follows:

```
lambda:  fct;
...
fct := p ::  "my_function";
```

where the `fct` variable has been declared as: "`lambda:  fct;`". Once this variable has been correctly initialized, it can be used to call the function like that:

```
x := fct ::  (y1,...,ym);
```

The suffix unary operator `::` calls and execute the function `my_function` inside the *sleeping* process `p`. The values y_1, ..., y_m denotes the arguments of the function. The situation for procedures is similar.

So the complete program could be:

```
program: p;
lambda: fct;
string: result;

begin
  p:=use("c:\\foo.oxo");
  fct:=p::"FormatDate";
  result:=fct::(13, "Feb", 1968);
  print(result);
end
```

This program is more concise and is strictly equivalent to the previous one:

```
begin
  print((use("c:\\foo.oxo")::"FormatDate")::(13, "Feb", 1968));
end
```

The second program is described in order to illustrate that `program` and `lambda` values can be used anywhere where such types are expected.

[1] This approach is near of the `C` philosophy, where functions can be used as procedures.

So far, one were visiting the calling program, but how does look the called program "`foo.v2`" ? Well, this program is as any other `V2` program. The only difference is that functions/procedures that can be called from outside must be preceded with the `export` keyword. So the `FormatDate` function would have the following syntax:

```
export function string FormatDate( integer:  d, string:  m,
integer:  y) ...
```

The syntax is the same for a procedure. The `export` keyword allows you to define functions whose the use is private and other ones whose the use is not limited.

Another important principle is the stack! The stack is preserved as long as it could be needed to execute the process or a function/procedure attached to this process.

Let us take look at these two programs

```
/********************/
/* calling program */
/********************/

program: p;
lambda: fct;

begin
  p:=use("c:\\foo.oxo");
  fct:=p::"nestor";
  p!();
  print(fct::(1));
  print(fct::(3));
end
```

and

```
/*********************************************/
/* called program. Stored in file c:\foo.oxo */
/*********************************************/
integer: n;

export function integer nestor(integer: a){
  n:=n+a;
  return n;
}

begin
  n:=1;
end
```

The execution of the first program will call the body of the `foo` program and put the 1 value into the global variable `n`. Next, the function `nestor` is called. This function will use the variable `n` as the previous execution let it in the stack. In our example, `n` will be worth 1 and the function will return 1+1! The second call will return 2+3.

Nothing prevents you to initialize two process from a program. Let us modify the calling program as:

```
program: p1,p2;
lambda: fct1,fct2;

begin
  p1:=use("c:\\foo.oxo");
  p2:=use("c:\\foo.oxo");
  fct1:=p1::"nestor";
  fct2:=p2::"nestor";
  p1!();
  p2!();
  print([fct1::(1),fct2::(3)]);
end
```

The execution of this program will print 2 and 4 as result. The execution of the first function will not influence the stack of the second function!

Now that one can call extern functions/procedures, let us examine how a program can retrieve the arguments and return a result when it is called from another program. In the following extract, the "`foo.oxo`" program is called two times. The first call tests the result of the program (that represents an error code) to display a message. The second call is more confident and does not test the error code.

```
program: prog;
begin
  prog:=use("c:\\foo.oxo");
  if prog!(1,2,3)=0 then {
    print("error message");
  }
  prog!(4,5);
end
```

Let us remark that the number of arguments may change from one call to another. The called program is defined as:

```
integer: sum,i;
begin
  if Length(Environment) then {
    for i in Environment do {
      sum:=sum+i;
    }
    print(sum);
    return 1;
  } else {
    return 0;
  }
end
```

We may observe two important changes from previous versions:

1. The global variable `Environment` is not defined in the program although it is used. This variable is predefined in `Voyager 2` and denotes a list. This variable is initialized with the list of the arguments present in the call. This variable can be used as any other variable.

2. The `return` instruction is now allowed in the body of the program. As expected, it returns a value to the calling program. The type of this value is not specified and can be of any type (`integer`, `string`, `list`, ...).

Some details were not fully explained in this part. The complete formal definition of each concept is given in the last section (section 16.4).

Another important aspect is that the user has not to care about the process unloading. As for strings and lists, the `Voyager 2` language automatically unloads process from the memory (image and stack) once a process is no more needed.

Because, `process` and `lambda` values are first order classes, you can manipulate these values as any other value. You can pass such expressions as arguments to functions (extern or not), store it into lists, ... Only IO output are not allowed: there is indeed no reason to print or read a process/value from a device (file/keyboard).

Sometimes, the process loading can fail (not enough memory, `.OXO` file corrupted, security failure, mistyped filename, ...). In such events, the loading mechanism returns either a `program` or a `lambda void` value that can be tested with the usual functions (`IsVoid` or `IsNoVoid`). The user has to explicitely test these values in insure the program correcteness.

16.3 Libraries

Libraries are just a convenient cosmetic ointment over the previous concepts. Because process are very often used as "libraries" than as real "process" (as known in operating system). The dynamic management of such process is too heavy and unnecessary. For this reason, `Voyager 2` has a special syntax to declare libraries.

Let us suppose that you have defined a program with several functions for advanced functionalities (for instance, to manage trees, associative lists, ...). To use all these functions, you should load this library/program each time, declare `lambda` variables for each used function/procedure and so on.

`Voyager 2` allows the declaration of a library at the beginning of the program. Here is an example:

```
/*************************/
/* libraries declaration */
/*************************/
use "c:\\lib\\tree.oxo" as mylib;
use mylib.DisplayTree as DisplayTree;
use mylib.ErrorProcess as TreeError;
use mylib.ComputeDepth as Depth;
use "c:\\lib\\assoc.oxo" as associative_list;
use associative_list.SetAssoc as SetAssoc;
/* Global Variables */
```

```
integer: a,b,c;
...
begin
  ...
  DisplayTree::(MyTree,File);
  print(Depth::(MyTree));
  ...
end
```

The `use` keyword denotes here two possible uses. The first one gives a logical name to a process/library. The second one gives a logical name to function/procedure inside a library.

As we already explained, libraries syntax is just a cosmetic layer on known concepts. Nevertheless, this allows to initialize process whatever the body is executed or not. This characteristic is important when you whish to use a library that needs another library. If the program needs the library `tree.oxo` and if this last one needs the library `record.oxo`, `record.oxo` must be loaded before any function/procedure of `tree.oxo` is called! And this is ensured by the `use` statement at the beginning of the program. Otherwise, you should execute the process first before executing any function from this process.

To make easier the programming job, the compiler produces a file with the extension `.IXI` that contains all the needed `use` declarations for each function/procedure defined with the `export` keyword. This file can be included with the directive explained in chapter 17.

16.4 Formal Definitions

16.4.1 The `use` Function

function program: p `use` (string: s)

Precondition. s is string that denotes the name of a `Voyager 2` program. The name must follow the syntax of paths in `MS-DOS`. This program must have the same security privileges as the calling program (see chapter 18).

Postcondition. p denotes a new process that has been loaded in memory with a newly created stack. The process is just loaded and no execution has been launched!
on error: The void value is returned.

16.4.2 The `!` suffix unary operator

Syntax: P `!` $(a_1, \ldots, a_n)$
P denotes a `program` expression. If P is not void, then the program/process denoted by P is called in using the $a_1, \ldots, a_n$ values as arguments. Otherwise, nothing happens. If the "!" operator is used inside an expression, then the called program must necessarily return a well-typed value as expected depending on the context. If it is used as an instruction, the returned valued is ignored.

16.4.3 The :: Suffix Unary Operator

Syntax: `F::(`$a_1, \ldots, a_n$`)`
F is any expression that is evaluated as a lambda value F'. Depending on the nature of F', the function/procedure denoted by this value (F') is called in using the expressions $a_1, \ldots, a_n$ as parameters. The number of arguments must be strictly the same as the number specified in the definition of the function/procedure. Because the compiler does not have enough information, it can not check if the number of arguments is correct! If F' denotes a function, the " F::($a_1, \ldots, a_n$)" expression will be evaluated as the result of this function. Otherwise, the procedure is just called. Once again, the compiler does not have enough information to check that functions are not used as procedures or reciprocally. If F' denotes the `void` value, then nothing happens and the execution is aborted with an error message. If there is a wrong number of arguments, the stack will be corrupted and the program will stop without any explicit messages! The same event occurs if a procedure is used as a function and reciprocally. Such errors are programming errors and must not be catched by the compiler/abstract machine. As for any function or procedure, arguments must have exactly the same types as the ones found in the definition.

16.4.4 The :: Binary Operator

Syntax: `P :: N`
P is a `program` expression and N is a `string` expression. The `::` operator is evaluated as a `lambda` expression that corresponds to a function/procedure named N and defined in the program P. The stack of P will be used during the execution of the function/procedure. Let us name R the result, then if `IsVoid(P)=TRUE` then R`=Void(_lambda)`. Otherwise if the N string does not match a function or a procedure in the specified program, then the `void` value is still returned.

16.5 Literate Programming

Programmers often document their program with comments. But few environment allow to recover the program documentation from comments[1]. `Voyager 2` tries to use comments present in your programs to document the `ixi` file produced by the compiler. In the `Voyager 2` syntax, "*explain clauses*" can occur in the head of the program and inside each function/procedure. Explain clauses are used by the compiler to produce documented `ixi` files. The figure 16.2 shows how the compiler works on a sample.

[1] "*Literate Programming*" first appeared in the `WEB` programming language that is a mix of TEX sentences and `Pascal` statements. `WEB` was defined by D. Knuth.

```
explain (*Factorial Library. Author: Nestor Burma *)

export function integer fact1(integer: a)
explain (*fact1 computes the factorial of its argument in using
a recursive algorithm *)
{ if a=0 then { return 1; }
  else { return a* fact1(a-1); }
}

export function integer fact2(integer: a)
explain (*fact2 computes the factorial of its argument in using
an iterative algorithm *)
  integer: i, result;
{ result:=1;
  for i in [1..a]
  do { result:=result*i; }
  return result;
}

begin
end
```

⇓

```
/* FILE GENERATED ON: 23/IX/1997 at 10:44,24 secs
** Please, does not modify this file.
** Voyager 2 Declarations
** Compiled with Version 3 Release 0 Level 2 */

use "facto.OXO" as facto;

/******************
 * Documentation: *
 ******************

Factorial Library. Author: Nestor Burma */

use facto.fact1 as fact1;

/* FUNCTION returns integer
   Arguments:
    1) integer: a
   EXPLAIN:
fact1 computes the factorial of its argument in using
a recursive algorithm */

use facto.fact2 as fact2;

/* FUNCTION returns integer
   Arguments:
    1) integer: a
   EXPLAIN:
fact1 computes the factorial of its argument in using
an iterative algorithm */

/* IXI file completed */
```

Figure 16.2: Literate Programming: The `ixi` file (bottom) is produced from the `Voyager 2` program (top). Each "explain clause" is used to document exported functions as well as the library itself.

Chapter 17

The Include Directive

Recurrent needs were observed during the programming phase. For instance, you always use the same constants, functions, procedures, ... For this reason, programmers had often to duplicate sections from one program into another program. Such programs become rapidly difficult to maintain.

It is now possible to include files into a program in using a directive statement. The syntax is the same as for the `C` language. Its syntax is: `#include "..."`. Spaces are not allowed between the `#` character and the `include`. However, the `#` may appear anywhere in a line and must not necessarily occur at the beginning of the line.

This directive may appear anywhere in the program: in the library section, in the global variables declarations, between statements, or even inside an expression. The semantics of this directive is quite simple: the compiler replaces the directive with the content of the file specified in argument. The backslash characters do not (and must not) need to be escaped. Here follows an example:
Example:

```
#include "c:\lib\tree.ixi";
...
integer: n;
#include "c:\lib\rtf_cst.h2";
...
function char foo(){
  ...
}

#include "c:\misc\error.h2";

begin
  #include "c:\misc\copyrigh.h2";
  ...
end
```

□

The language does not force the extension names. However, we stringly recommend the use of `.ixi` for libraries declarations and `.h2` for other files. The `.h2` extension should be used for any file that you reserve for inclusion

purposes. Although you can choose another extension name, life would be easier if everyone respects this convention (as in `C`).

We also recommend to avoid as much as possible[1] the use of this directive. Libraries can often be used in place of this directive and should be prefered for methodological reasons.

If an error occurs in an included file, then the compilation process fails as for any other reason. However, if the compiler fails in the opening of an included file, this one is simply skipped and the compiler produces a warning. Very often, this will cause other errors but this will sometimes "spare" your life!

[1] The `include` directive really includes the content of a file and thus, enlarges the `.oxo` file size. Since the size of `.oxo` files is limited, you could exceed this limit.

Chapter 18

Security

As registered user, you should have received an electronic key (EK). This one stores informations in a crypted memory:

The user ID: A unique number associated with each user. A company having several EK's will share the same ID.

The key ID: A unique number associated with each EK. Two EK's have distinct key ID's.

Compile capability: If this flag is `true`, then the user can use the Voyager compiler. Otherwise, the compiler will stop its execution with an explicit error message.

The distribution capability: If this flag is `true`, then you can develop Voyager programs for other users and restrict its use either for a user, or for an EK. This capability is mainly interesting for companies selling the products from the `DB-MAIN` Research Group.

It is impossible to run the compiler without the EK, and the use of the compiler with the EK is restricted as explained in the above chapter.

When the "Compile capability" is set, the compiler produces `.OXO` files that are restricted on computers having an EK with the same "User ID" as the one found on the computer on wich the compilation was executed.

When the "Distribution capability" is set, the user can specify an explicit "user ID" or an explicit "EK ID" that restricts the use of the `.OXO` file respectively on a `DB-MAIN` process having the key with the ad-hoc information.

With the "Distribution capability" enabled, you can use additional switches on the command line of the compiler:

`-Kclient:` to specify the user ID (*cfr.* `-infokey`).

`-Kkey:` to specify the key ID (*cfr.* `-infokey`).

`-Kall:` to remove the restrictions on the distribution of the `.OXO` file.

`-infokey` ***nnn*`:`** to specify the ID used by the compiler (*cfr.* `-Kclient`,`-Kkey`). ***nnn*** is the number/ID information.

Example:

Here follow some examples:

```
comp_V2 foo.v2 -Kclient -infokey 31414
comp_V2 foo.v2 -Kkey -infokey 27182
comp_V2 foo.v2 -Kall
```

the command line "`comp_V2 foo.v2`" is the same as the command line "`comp_V2 foo.v2 -Kclient -infokey` x" where x is the user ID of your electronic key.

□

Part IV

Appendix

Appendix A

The `Voyager 2` Abstract Syntax

This chapter gives the abstract syntax of the `Voyager 2` language. The syntax is defined as a set of rules written in extended BNF. The following conventions are respected.

- **head** ← body: A rule.
- "`example`": literal characters.
- `example`: A keyword.
- *example*: A terminal word that represents a class of tokens. (*cfr.* 2 for more details)
- **example**: A non terminal word. This one must be defined in a rule.
- the | operator denotes a disjunction in a body.
- $\langle$example$\rangle_{0,\infty}$: Example is repeated 0 or more times.
- $\langle$example$\rangle^{\alpha}_{0,\infty}$: Example is repeated 0 or more times and items are separated by "α".
- $\langle$example$\rangle_{1,\infty}$: Example is repeated 1 or more times.
- $\langle$example$\rangle^{\alpha}_{1,\infty}$: Example is repeated 1 or more times and items are separated by "α".
- $\emptyset$: the empty word.

The Syntax

program ← **explain-clause** $\langle$**use-clause**$\rangle_{0,\infty}$ $\langle$**def-var**$\rangle_{0,\infty}$ $\langle$**def-fct**$\rangle_{0,\infty}$ **body**

use-clause ← `use` *string* `as` *identifier* "`;`"
| `use` *identifier* "`.`" *identifier* `as` *identifier* "`;`"

def-var ← *type* ":" ⟨**one-var**⟩$^{,}_{1,\infty}$ ";"

one-var ← *identifier* ⟨"=" **expr**⟩$_{0,1}$

def-fct ← **def-function** |**def-procedure**

def-function ← ⟨export⟩$_{0,1}$ function *type identifier* "(" ⟨**arg**⟩$^{,}_{0,\infty}$ ")" **explain-clause** ⟨**def-var**⟩$_{0,\infty}$ "{" ⟨**instr**⟩$_{0,\infty}$ "}"

def-procedure ← ⟨export⟩$_{0,1}$ procedure *identifier* "(" ⟨**arg**⟩$^{,}_{0,\infty}$ ")" **explain-clause** ⟨**def-var**⟩$_{0,\infty}$ "{" ⟨**instr**⟩$_{0,\infty}$ "}"

arg ← *type* ":" *identifier*

body ← begin ⟨**instr**⟩$_{0,\infty}$ end

instr ← ∅
| **designer** ":=" **expr** ";"
| goto *identifier* ";"
| continue ";"
| break ";"
| halt ";"
| label *identifier* ";"
| return ⟨**expr**⟩$_{0,1}$ ";"
| **attach-stmt** ";"
| **move-stmt** ";"
| **addcursor** ";"
| **setval** ";"
| **dynamic-call** ";"
| **loop-stmt** ⟨";"⟩$_{0,1}$
| **ifthenelse** ⟨";"⟩$_{0,1}$
| **while-stmt** ⟨";"⟩$_{0,1}$
| **switch-stmt** ⟨";"⟩$_{0,1}$
| **repeat-stmt** ";"
| **call-procedure** ";"

omega2 ← "-" |"+" |"*" |"<" |">" |"<=" |">=" |"=" |"<>" |"++" |"**" |and |or |xor |mod

expr ← **expr omega2 expr**
| designer ":==" **expr**
| "-" **expr**
| not "(" **expr** ")"
| "(" **expr** ")"
| "[" ⟨**expr**⟩$^{,}_{0,\infty}$ "]"
| "[" **expr** ".." **expr** "]"
| **expr** "[" **designer** "]" "{" **constraint** "}"
| use "(" **expr** ")"
| **expr** "::" "(" ⟨**expr**⟩$^{,}_{0,\infty}$ ")"

| **expr** "`::`" **expr**
| **expr** "`!`" "`(`" ⟨**expr**⟩$^{;}_{0,\infty}$ "`)`"
| **designer**
| *integer*
| *char*
| *string*
| *file*
| **call-procedure**
| **create-inst**

constraint ← **expr**
| **expr** "`:`" **expr** ⟨ `with` **expr**⟩$_{0,1}$

setval ← **expr** "`<--`" **expr**

dynamic-call ← **expr** "`::`" "`(`" ⟨**expr**⟩$^{;}_{0,\infty}$ "`)`"
| **expr** "`!`" "`(`" ⟨**expr**⟩$^{;}_{0,\infty}$ "`)`"

loop-stmt ← `for` **designer** `in` **expr** `do` "{" ⟨**instr**⟩$_{0,\infty}$ "}"

ifthenelse ← `if` **expr** `then` "{" ⟨**instr** ⟩$_{0,\infty}$ "}" ⟨`else` "{" ⟨**instr**⟩$_{0,\infty}$ "}" ⟩$_{0,1}$

switch-stmt ← `switch` "`(`" **designer** "`)`" "{" ⟨**case-stmt**⟩$_{0,\infty}$ **default-case** "}"

case-stmt ← `case` **expr** "`:`" ⟨instr⟩$_{0,\infty}$

default-case ← ⟨`otherwise` "`:`" ⟨**instr**⟩$_{0,\infty}$⟩$_{0,1}$

repeat-stmt ← `repeat` "{" ⟨**instr**⟩$_{0,\infty}$ "}" `until` **expr**

while-stmt ← `while` **expr** `do` "{" ⟨**instr**⟩$_{0,\infty}$ "}"

⟨**expr**⟩$_{0,1}$ ← ⟨**expr**⟩$_{0,1}$

call-procedure ← *identifier* "`(`" ⟨**expr**⟩$^{;}_{0,\infty}$ "`)`"

attach ← `attach` *identifier* `to` **expr**

move-stmt ← *identifier* {"`<<`" |"`>>`" }⟨**expr**⟩$_{0,1}$

create-inst ← `create` "`(`" **expr** "`,`" ⟨**simple-field**⟩$^{;}_{0,\infty}$ "`)`"

simple-field ← **expr** "`:`" **expr**

addcursor ← **expr** {"`<+`" |"`+>`" }**expr**

designer ← *identifier*
| *identifier* "`.`" **expr**

explain-clause ← ⟨`explain` "`(*`" *text* "`*)`"⟩$_{0,1}$

Remarks

The `#include` directive does not appear in the `Voyager 2` syntax since it is replaced anywhere by the content of the specified file.

Appendix B

The VAM Architecture

Although the only tool you see is the compiler, it is worthwhile to know that `Voyager 2` is interpreted. In fact, there are two languages: `Voyager 2` (V2) and `Voyager 1` (V1). The first one was described in this manual. The second one, V1, is just an intermediate level between V2 and the abstract machine: the VAM[1].

The picture of figure B.1 shows how a V2 program is translated to a binary file (`prog.oxo`) that can be loaded directly into `DB-MAIN` in order to execute its task. The following table shows extracts from three files: `prog.v2`, `prog.v1` and `prog.oxo`:

prog.v2	prog.v1	prog.oxo
begin	push-int 1	13/1
print(1+2);	push-int 2	13/2
end	add	65
	print	43

The `prog.oxo` is just a binary file composed of 6 words (16 bits): 13,1,13,2,65 and 43.

V1 looks like an assembler language and the "oxo" file is just a binary translation of each instruction with its operands. The "oxo" file can be fastly loaded into the `DB-MAIN` tool since the parsing has already been done. The general architecture of all these processes is depicted in figure B.1. The compiler you used is represented by a box that contains two hidden processes: the real V2 compiler (named `comp_V2'`) and the V1 compiler (named `comp_v1`). What you see is just the translation of the V2 program into the binary file. Once this compilation is completed (and there was no errors!), this program can be loaded into the tool and can be executed.

[1] **V**oyager **A**bstract **M**achine.

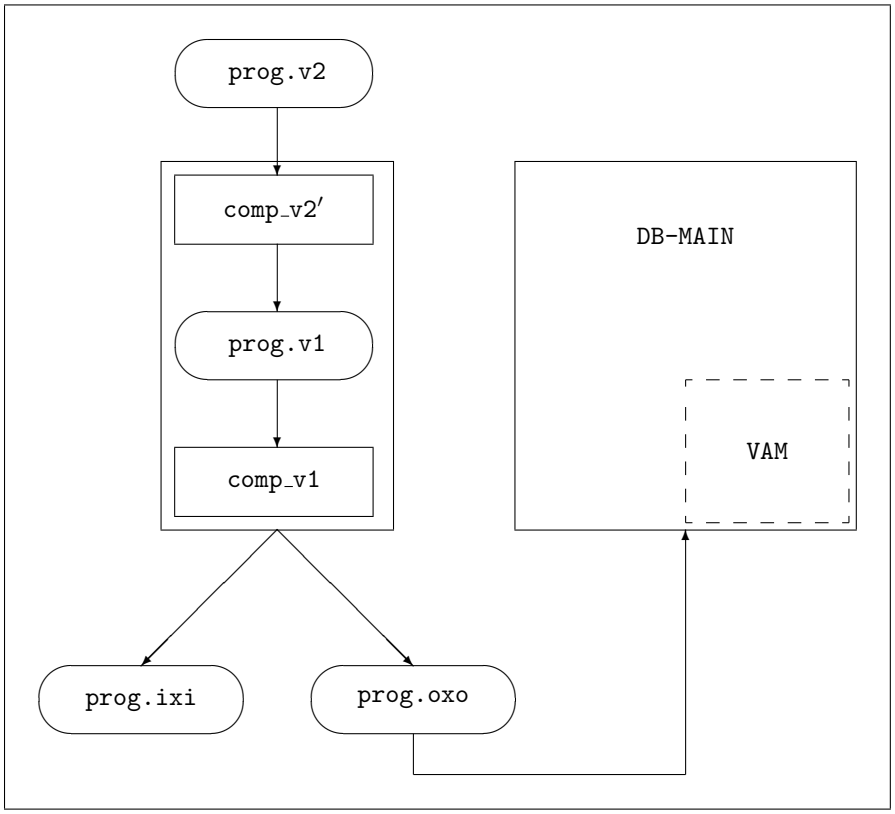


Figure B.1: The Voyager Architecture.

Appendix C

Error Messages while Compiling

The compiler produces three types of error messages during the compilation:

warning: The error is not important and the compiler is smart enough to continue in producing the right code. Exemple: a `return` instruction is followed by an expression in a procedure.

error: The error prevents the compiler from generating the right code. The compiler will skip the error and continue its job looking for other errors. The code is not produced.

fatal: The error is too important to continue the compilation. There is no code produced of course.

All the errors produced by the compiler are documented here after. The compiler always tries to display the line number of the error. Sometimes this number does not correspond to the right line. It is the case for composed instructions. This is a known bug[1].

1: `Buffer to small ! (maybe a line with too much characters)`

A line in your program is too long for the buffer of the compiler. It should be possible to split it in several lines.

2: `Parsing error !`

There is a syntax error. The compiler should show the content of the line as well as the line number where the error occurred. Note the character '‾' that indicates where the syntax is not verified. Very often, it lacks a ';' separator in the previous line!

3: `Error in opening file !`

[1] We could say: it is not a bug but a feature. Indeed, the compiler only knows the line number when one instruction is fully parsed. For this reason, the line number corresponds to the last line of the instruction and not the first one.

It is impossible to open the specified file (.V2) or to create the output file (.V1). Is it shared by other applications? Does it exist? Is your path valid?

4: `One function already exists with the same name !`

You are defining a function/procedure twice.

5: `The left hand expression of an assignment must be a variable or a field !`

Your program has one instruction $E := T$; where E is not a variable. Only variables (or with a field) are admitted in the left part of assignments.

6: `This procedure is used but not defined !`

One instruction is calling a procedure that is not defined. Note that lower and upper case characters are considered as distinct in `Voyager 2`.

7: `Unknown type !`

Your are using a bad type. Probably a misspelled type like `integger` or `siattribute`.

8: `This identifier is neither a variable or a constant !`

A variable is expected here and your identifier was not defined as a variable.

9: `The procedure is used here as a function !`

You are using a procedure as a function. There is a problem in your analysis.

10: `A variable is expected here !`

You have specified a field with a variable used as loop-variable in a request. Only variables are allowed here. Example: `ENTITY_TYPE[e]{...}`.

11: `This identifier is already defined !`

You are defining an identifier twice. Functions/procedures names, global variables, parameters names and local variables must be all different.

12: `Incorrect number of arguments !`

You are using less or more arguments than required in the definition of a function/procedure.

13: `A field is not allowed with a constant name !`

You are using a constant name where a variable is expected.

14: `This identifier is neither a function nor a buil-tin`

You are using an undefined function. Check the spelling.

15: `Only a variable can be passed by address !`

You are passing a value to a built-in procedure/function that expects a variable passed by reference.

16: `This local variable is already defined !`

There is a conflict between your local variable and either a global variable, a constant name or a function/procedure name.

17: `Local constants not yet supported !`

This feature is not yet implemented although the parser understands it. Note that the feature is not explained in this reference manual.

18: `Return statements in functions must have an expression !`

You are using the `return` instruction inside a function with no value. Note that the returned value must have the same type as the one specified in the head of the function.

19: `Return statements in procedures can not have expression !`

You have specified a value after the `return` instruction inside a procedure. This value has no sense and is not allowed in `Voyager 2`.

20: `Return statements in the body of the program can not have an expression !`

You have specified a value after the `return` instruction inside the main body (the part between the `begin` and `end` keywords). This value has no sense and is no allowed in `Voyager 2`.

21: `"continue" statement not allowed here !`

The `continue` instruction is only allowed inside `while`, `repeat` and `for` instructions.

22: `"break" statement not allowed here !`

The `break` instruction is only allowed inside `while`, `repeat` and `for` instructions.

23: `The first character of identifiers must be different of '_' !`

The '_' character is used for reserved keywords.

24: `Internal Error !`

The compiler has reached a dangerous state. Please warn the DB-MAIN research group.

25: `Not enough memory !`

Not enough memory to allocate dynamic objects. Please close some applications or restarts Windows.

26: `Unterminated string !`

You have omitted to close a string. Add a double quote at the end.

27: `This word is a reserved keyword !`

Change the spelling of this identifier to avoid the conflict.

28: `procedure expected in a call !`

You are using a function where a procedure is expected.

29: `Sub-process can not be executed !`

The compiler can not run a sub-process. The `.oxo` file has not been produced. You have too many applications, you do not have enough memory or the process was not found. Its name is `comp_v1.exe`. Check that this file is correctly installed. A more precise message should have been displayed indicating the exact problem.

30: `A */ is probably missing`

A comment is not terminated or is too long.

31: `A *) is probably missing`

An explain clause is not terminated or is too long.

32: `A literal string is expected here!`

The expression must be a string (ie. "...).

33: `Fields are not allowed for iterator variables in loops.`

Expressions of the form *variable.field* are not allowed in the loop statements (`for-in-do`).

Let us note that the semantics of `Voyager 2` prevents the compiler to trap some errors. The main reason is that `Voyager 2` is weakly typed and the type verification is sometimes impossible. For this reason, it is possible to compile strange programs and to execute them. Fortunately, the VAM will stop them.

For instance: the compiler will compile the next program without any errors or warnings:

Example:

```
file:  F;
begin
    F:="Hello"+(5++'i');
    print(F);
end
```

□

34: `The identifier does not denote a ‘‘program’’. Define it in a use clause before`

The program uses an identifier in a “use” clause that was not declared as a library in a previous “use” clause.

35: `Include directive skipped. Impossible to open the file.`

The filename specified in an include directive does not denote a file or this file can not be opened.

Appendix D

Error Messages during the Execution

The VAM (**V**oyager **A**bstract **M**achine) is able to trap most of the problems that can occur during the execution. Each time it is possible, an error message is displayed indicating the cause of the problem. Unfortunately, it is impossible to trap all the possible errors. This means that wrong voyager programs may get the VAM wrong. The error may be in your program or in the VAM. Our experience is that the error is often in the voyager program.

1: `Another type is expected.`

The type of an operand does not match the expected type of an operation.

2: `Internal Error: please stop here.`

This error should not appear. Please report it to the development team with all the available informations.

3: `You are using an invalid field.`

The used field is not valid for this type of object. Example: `print(dto.identifier);` where `dto` is a `data_object`.

4: `No enough memory for allocation.`

It is impossible to have more memory for dynamic data. You may close some applications or restart Windows.

5: `A list is expected here.`

You are passing a non-list value to an operation that expects a list.

6: `No program loaded.`

This error is obsolete.

7: `Impossible to open file.`

Impossible to open the `.oxo` file. Do you have compiled it?

8: `Illegal Instruction`

The `.oxo` file is probably corrupted. If it is the case, then recompile it. If it does not work, then the memory is corrupted, restart Windows. If this fails again, then warn the development team.

9: `The instruction (yet valid) is not defined so far.`

This message should never appear. Warn the `DB-MAIN` development team.

10: `Cell must be active !`

You are referencing an invalid cell. This message will occur during the following program:

```
cursor: c; ...; kill(c); print(get(c)); ← error 10.
```

This is just an example.

11: `The list can not be empty !`

The expected argument should be a list with at least one active cell.

12: `The file is corrupted`

The `.oxo` file is corrupted. Please recompile your program.

13: `Error while reading the file`

Your `.oxo` file is corrupted. Recompile your program. This may occur when you are using old versions of `.oxo files with a more recent version of DB-MAIN.`

14: `Argument of SetPrintList too large for the buffer !`

Arguments of the `SetPrintList` are stored in static strings and therefore the size is limited. Remove some characters.

15: `This type can not be read/written !`

You are printing/reading an invalid type of data. For instance, you can not print values of type `list` or reference to objects like `data_object`.

16: `Invalid field in Create operation !`

You have specified an invalid field in the `create` instruction.

17: `A mandatory field is lacking in Create operation !`

Check the `create` instruction, all the mandatory fields must be present.

18: `Integrity rules are not checked !`

A `create` instruction has violated an integrity rule during the execution. Read the documentation of `create` for more details about the integrity rules.

19: `The buffer of the lexical analyzer is to small`

You are trying to parse a too big piece of text for the buffer of the lexical analyzer. Consult the `MAX_LEX_BUFFER` for more details on the size of this buffer.

20: `Several choices are identical in MakeChoice, suppress them!`

Several arguments of the `MakeChoice` statement are identical. It is probably a typing error.

21: `A semi-formal field in the text is corrupted!`

The syntax of a textual property is invalid.

22: `The dynamical property does not exist!`

You are referencing a dynamical property that is not referenced as one instance of the `meta_property` object type.

23: `Remove is not allowed for this type!`

The `remove` instruction must be used to remove objects of the repository. The type of the value you are using as argument of this instruction is invalid!

24: `Impossible to start the AsbtractMachine with this program`

You have specified a wrong name/path of a Voyager program. Other causes can be a damaged file, or a problem with dynamic ressources (file handle/memory/...).

25: `Bad lambda expression`

Your *lambda* expression is invalid. You are problaby trying to use a function that does not exist or that is mispelled.

Appendix E

Frequently Asked Questions

E.1 Environment Relation Questions

1. How do I compile a program?

Once you have saved your program in a file (with your favorite text editor), say "`prog.v2`", you can compile it by using the `Voyager 2` compiler.

The compiler is a 32-bits application that can be executed from the DOS prompt. If your program is correct, the compiler has produced a file called "`prog.oxo`. This file can then be loaded in the `DB-MAIN` tool. Otherwise, the compiler stops and prints all the error messages. Each time an error is encountered, an error message is displayed. After each message, the compiler expects a character. The effect of the entered character is:

`c`: continue and do not stop any more

`s`: stop now!

other characters: continue and stop on the following message

More options are allowed on the command line of the compiler and they are described here after:

syntax

`comp_v2 <`*`option`*`>`$\star$

where *option* can be

filename: any string with no space character and with the first character different from the character '-' can be a filename. By convention: the first filename found in the list is the name of the input file . The second filename is the output file. This filename must not have any extension. The compiler adds automatically the `.oxo` extension. If the output file is not specified then the name is deduced from the input file.

-date: prints the version of the compiler and stops immediately.

If the filename is missing in the list of options, the compiler will ask it during its execution.

---

2. How do I write efficient programs ?

`Voyager 2` was not built to be efficient but to write programs rapidly and easily. For this reason, it is bad idea to try to write a number crunching program in `Voyager 2` ! But there are some recipes to make your programs a little faster:

- Avoid lists in the `print`x instructions: prefer `print(x); print(y);` to `print([x,y]);`.
- Expand requests rather than using intermediate lists.

```
l:=request;
for x in l do {
⋮
}
```

```
for x in request do {
⋮
}
```

The right program will be faster than the left one.

- Use available built-in instructions when it is possible: prefer a `for-in-do` to a `while` instruction.
- In the `for-in-do` instruction: when the list looks like $[a..z]$, the z expression is evaluated at each loop. Thus if this expression is quite complex, you should evaluate it before the loop and save the result into a variable.

---

3. Why do I have two windows after the compilation?

The compiler needs to launch another compiler during its execution. The Windows library used to display the error messages does not allow to close these windows once the program halts. You just have to close them yourself! This problem does not occur anyway with the 32bits version.

---

4. I can not close the console ! Why?

The console is locked until the $\check{2}$ program is finished.

---

5. When I load program, `DB-MAIN` tells me that the version of the program is too old!

Although DB-MAIN is backward compatible with all the versions of the language, the binary format may change from one version to another one. When you get this message, just recompile the .V2 file with the ad-hoc compiler.

---

6. Why does the compiler find errors in my program although it was working fine with the former version?

It means proprably that some identifiers used as variable/constant/function name are now reserved keywords in the new version. It is especially true for "dot-expression" (*variable.expression*). In the former version, the compiler was able to distinct a variable declared as `integer: sem` from a field used in a *dot-expression* like `sch.sem`. Because right-hande-side expression of the "." operator may now be any expression, the compiler is no more able to distinct them. So rename your variables to avoid the clash. The error message should be enough precise to fix the problem yourself.

E.2 Language Specific Questions

1. In a predicative query, DB-MAIN tells me that there is an invalid assignment. Why?

Well, it is possible that you are using a sub-type where a super-type is expected. Get a look at the following query:

```
ENTITY_TYPE[ent]{@SCH_DATA:[sch] with
      GetType(ent)=ENTITY_TYPE}
```

Although the list returned by the query will be composed of only entity types, the `ent` variable will be used as iterator in the generated code to find all the instances of data objects linked to the `sch` schema. These objects can be entity types, but also rel-types and attributes. Thus: it is possible that the VAM tries to put one attribute in the `ent` variable that should be defined as `entity_type`. Thus the correct query would be:

```
DATA_OBJECT[dto]{@SCH_DATA:[sch] with
      GetType(ent)=ENTITY_TYPE}
```

---

2. Is there a `nil` value like in `Pascal`?

No. The reason is very simple: there are no pointers in 2̌. However for references, there are special values denoted by `void`. To obtain the `void` value of the `entity_type`, one can use the function `Void(ENTITY_TYPE)`. This function can only be called for references and not for other types like: `integer` or `char`.

---

3. Why is my request looping?

It is prohibited to modify the value of a variable used as iterator in a query. For instance, this program is wrong:

Example:

```
data_object:  dto;
schema:  sch;
begin
    sch:=GetCurrentSchema();
    for dto in DATA_OBJECT[dto]{@SCH_DATA:[sch]} do {
       dto:=Void(DATA_OBJECT);
    }
end
```

□

---

4. How may I empty a list L?

Very simple: `L:=[1,2]; L:=[];`. The last instruction will empty the list L.

---

5. How may I test if a list is empty ?

Well, there is at least two simple ways to proceed. The first one is in matching the candidate list against the empty list: `if mylist=[] then ...`. If you do not like this first solution, you can also test the length of the list: `if Length(mylist)=0 then ...`.

Appendix F

Regular Expressions

A *regular expression* is a pattern description using a "meta" language. The characters that form regular expressions are:

`.`	Matches any single character.
`*`	Matches 0 or more copies of the preceding expression.
+	Matches 1 or more copies of the preceding expression.
`[...]`	Matches any character within the backets.
`?`	Matches 0 or 1 occurence of the previous expression.
`"..."`	Matches exactly the content enclosed between quotes.
x`..`y	Is a notation for a character range, e.g., "[0..4]" means "[0,1,2,3,4]".
`\t,\n,`$\backslash x$	Denotes the tabular, the newline characters and the x character when this one is already used by the regular expression language ([].*+...).

For instance `[a..zA..Z][a..zA..Z0..9]*` denotes the syntax of identifiers in Pascal and `[0..9]+[+-]?[0..9]+` denotes the syntax of real numbers (12,012,19.021,...).

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilateurs. Principes, techniques et outils*. InterEditions, 1989.

[2] A. Church. *The calculi of lambda conversion*. Princeton University Press, 1941.

[3] V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. Description du méta-schéma de l'atelier logiciel `DB-MAIN` version 1.0. Technical report, F.U.N.D.P., 1995.

[4] A. J. Field and P. G. Harrison. *Functional Programming*. International computer science series. Addison-Wesley, 1989.

[5] K. Jensen and N. Wirth. *Pascal. Manuel de l'utilisateur*. Eyrolles, 1978.

[6] D. E. Knuth. *The TEXbook*. Addison-Wesley, 1990.

[7] D. Roland. MDL: Programmer's guide. Technical report, F.U.N.D.P., 2000.

[8] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1991.

[9] R. Wilhelm and D. Maurer. *Les compilateurs: théorie, construction, génération*. Masson, 1994.

Index

D

E

F

G

S